ADH, Aspect Described Hardware-Description-Language

by

Su-Hyun Park

A Thesis

submitted in partial fulfilment

of the requirements for the degree

of

Master of Engineering

in Electrical and Electronic Engineering

in the

University of Canterbury

Signal and Image Processing Research Group
Department of Electrical and Computer Engineering
University of Canterbury

March 2006

Abstract

Currently, many machine vision, signal and image processing problems are solved on personal computers due to the low cost involved in these computers and the many excellent software tools that exist, such as MATLAB. However, computationally expensive tasks require the use of multi-processor computers that are expensive and difficult to use efficiently due to communications between the processors. In these cases, FPGAs (Field Programmable Gate Arrays) are the best choice but they are not as widely used because of lack of experience in using these devices, difficulties with algorithmic translation and immaturity of the design and implementation tools for FPGAs.

Programming languages are always evolving and the programming languages for microprocessors have evolved significantly, from functional and procedural languages to object-oriented languages. Nowadays, a new paradigm called aspect-oriented software development (AOSD) is becoming more widespread. However, hardware programming languages have not evolved to the same extent as the software programming languages for microprocessors. They are still dominated by the technologies developed in 1980s, which have significant deficiencies described in this thesis. Recent advances in HDLs (Hardware Description Languages) have taken a conservative approach based on well-proven software techniques.

Here, a new hardware programming language called ADH (Aspect Described Hardware-Description-Language) has been designed. This language is based on the AOSD technique, which is a recent technique used in software programming languages for microprocessors and is still largely in the research area rather than the mainstream. Hence, we are taking a big step forward in hardware programming languages. This language is specifically targeting the control modelling problems in addition to the signal and image processing domain problems. A compiler is implemented that produces VHDL codes running on FPGAs to support the designed language. Therefore, this thesis describes the designed language, ADH, and its compiler implementation in detail.

Acknowledgements

I would like to thank my supervisor Dr. Andrew Bainbridge-Smith, not only for his teaching, discussions and criticisms, but also for the support and suggestions that he gave me on other things as well. I would also like to thank my friends and colleagues for their inspiration, and the University of Canterbury where I have spent my undergraduate years as well for both B.E(Hons) and B.Sc. I shall always miss the lecture theaters, labs and my office. Special thanks also go to the Academic Skills Centre at the University of Canterbury which helped me a lot in creating a quality piece of writing.

Finally, but most importantly, I offer my heartfelt thanks to my parents, my sister and my girl-friend for their encouragement, love and support. Without any one of them, this work would not have been possible.

Contents

Chapter 1

Introduction

1.1 Usage of FPGAs

The Safe-T-Cam System from CSIRO developed for the New South Wales (NSW-Australia) Road Traffic Authority was developed in the early to mid 1990s [5]. This system photographs heavy transport vehicles at selected points on the NSW highway network and automatically locates the licence plate in the image, reads it using OCR (Optical Character Recognition), and transmits the results back to a central site so that these vehicles can be tracked. The triggering of the acquisition camera is initiated by a separate vision system that monitors a video stream of the highway and identifies trucks from other vehicle classes. When first implemented, this system was considered a high data-rate vision system and no single or dual processor solution could be used to identify and trigger the acquisition in real time. A single FPGA solution was developed to perform the processing [23]. The system has been in operation for about 10 years and has been rolled out to 21 sites [4].

An algorithms research group at the University of Canterbury has presented BSA (Brightest Spot Algorithm) based on the maximum subarray problem [7,8]. It works well for the smaller images but it is still unable to perform real-time operation of the larger (1024×768 *pixels*) images that are commonly used these days. Figure 1.1 shows the use of post-processed BSA algorithm on a PC (Personal Computer) that finds the brightest spots on a map. An FPGA based solution is under development to be able to operate in real-time.

Figure 1.1: The use of BSA algorithm on a map

It is common to see USB [12] interfaced "webcams" and Firewire [3,46] interfaced digital cameras. These are very popular choices and interfacing with PCs allow the capability of solving many image processing problems. However, these are not advisable for many industry projects because they often use "lossy compression techniques", particularly JPEG and MPEG, to reduce the communication transfer times over the limited bandwidth provided by both USB and Firewire interfaces. When reconstructed, these compressed images have lost resolution and compression algorithm can be considered to have injected noise or artifacts into the images that often strongly interfere with the performance of many machine vision algorithms. There are industry cameras [56] that can take larger (1280×1024 *pixels*) images at high frame rate (Up to 500 *frames/sec*) that produce enormous data rates (660 *MB/sec*). This data rate cannot be processed in real-time with single microprocessor solutions; therefore, they must be stored first and then post-processed.

The following two examples illustrate problems for which an FPGA solution would be well suited. The first one is another CSIRO project, RoadCrack, which images the road pavement looking for cracking as a means of road asset management [27]. The camera system is mounted to a truck that moves at highway speeds ($100km/hr$) and is able to detect 1mm cracks for the full truck width. In a typical day, $600km$ of road can be surveyed for a total camera data volume of about $360GB$. This must be processed in real time to compress the data, finally producing a summary report that fits on a $1.4MB$ floppy disk. Currently, thirteen Alpha processing

units onboard the truck are used to accomplish this task. Clearly, specially engineered solutions will be required to miniaturize the $16MPixel/s$ RoadCrack system to fit within a 4WD vehicle or small trailer. The Machine Vision group at CSIRO has developed an FPGA based computer board, called Hymod, specially for this type of application [9].

The second example is centered around the use of $500fps$ image sensors from PhotoBit [45]. This sensor is being looked at by astronomical imaging groups at both the Australian Defence Force Academy (ADFA) and the University of Canterbury. Both groups are interested in imaging atmospheric turbulence in order to better model its effects and for the use in adaptive optic systems. The PhotoBit sensor is capable of producing $650MPixel/s$ of image data. Researchers at ADFA have devised an interesting system that is able to spool data from this sensor onto a set of disks [55]. Their system, however, has little headroom for processing this data on the fly and it must therefore be post-processed.

1.2 Compute intensive problems

This project was brought about by the problems that could not be solved with single microprocessor solutions. The problems that we are interested in are machine vision, signal processing and control modelling. The computing devices available to tackle such problems include (1) single microprocessors, (2) multi-microprocessors, (3) dedicated ASIC (Application-Specific Integrated Circuit) hardware and (4) FPGAs (Field Programmable Gate Arrays). In choosing the devices, some trade-offs need to be made between performance, flexibility, cost, portability, power-demand, etc. and these are the factors that need to be considered.

We will briefly go through each of them, discussing their advantages and disadvantages. Of the four, microprocessors are relatively cheap compared to the others and they are very flexible. A lot of applications can be operated on microprocessors. For example, MATLAB [47] is an excellent software tool that can be used to simulate the problems on a PC. The solutions provided by MATLAB are typically implemented on PCs in other programming languages, for example *C*, due to the

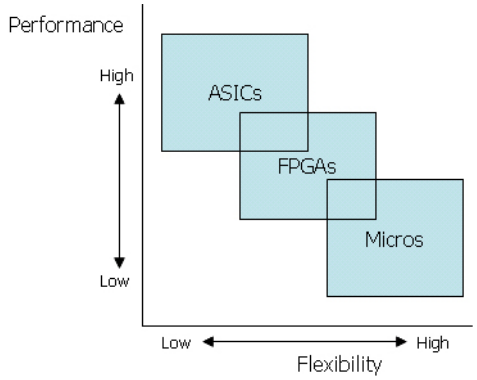


Figure 1.2: The performance versus flexibility for the different computing devices

low performance of MATLAB. However, employing PCs is not always possible due to their size, weight and power demands. As the microprocessors are designed for general purposes, their performance is the lowest of the four.

ASIC has the best performance of the four but as its name suggests, it is targeted for a specific problem only and there is no flexibility. The cost associated with ASIC is also very high unless mass produced. The nature of problems in the signal and image processing domain are often experimental; hence, the solutions to the problems might change over time. In this case, employing ASIC is an expensive way due to the high-cost in the designing and production of dedicated hardware devices.

FPGAs are sort of in between microprocessors and ASIC; see Figure 1.2. Since they are reconfigurable, they have room for flexibility and perform better than microprocessors. As stated earlier, the solutions to the problems can change over time; therefore, the factors that we consider the most in this thesis are the performance and the flexibility of the given applications — signal and image processing. Hence, the best choice is the FPGAs.

FPGAs are becoming a more popular choice for digital signal processing applications ahead of microprocessor-based solutions in a number of specialist areas. FPGAs looks similar to microcontrollers at first glance, as both are programmed to perform a task. In the past, microcontrollers have been developed on the basis of 'state machine'. There are sets of instructions and registers to allocate and store the values to be processed and they are done in a sequential order. However, FPGA programs must be compiled into hardware description code where there is

no predetermined control or execution unit, which allows power saving and massive parallelism. They offer performance advantages in restricted energy demand, size and weight. Their enhanced popularity is due to larger logic capacity and capability, which allows them to implement substantial parallelised signal processing algorithms in hardware; see Chapter 2.1.

1.3 Motivation

FPGAs are called reconfigurable because they are programmable and thus effectively provide reprogrammable hardware. The only drawback is in the way they are configuring these devices. The configuration is typically done in schematic diagrams or HDLs (Hardware Description Languages). There are not many HDLs available and the most dominant languages, VHDL and Verilog, have significant deficiencies; see Chapter 2.2.4. The designers are required to focus more on implementation issues rather than on abstracting the conceptual level; see abstraction problem on page 14. The technology of electronics (including microprocessors and FPGAs) and software engineering have evolved, which has led programming languages for microprocessors to evolve; however, the programming of FPGAs have not kept pace. We are still using the languages developed in the 1980s (VHDL, further explained in Chapter 2.2.1).

There are other advanced HDLs such as SystemC [24] and HandleC [15], which have been out for only a few years and still have a lot of limitations. As their names suggest, they are developed based on the *C* language, which is a well-known general purpose programming language for microprocessors. They provide a familiar starting point for new users and are less target specific. However, they started as simulation languages and tools rather than device generation tools. The target machines that the device compilers can generate are very limited. These languages still do not use the up-to-date technologies in software engineering such as AOSD (Aspect-Oriented Software Development), which is explained in Chapter 2.3. There are also different but related research for HDL such as JHDL [11,44] and *Java* based API (Application Program Interface) called JBits [34,35], and a combination of the

two called JHDLBits [62].

Therefore, we propose a new language and a compiler to program these devices, with a working name of ADH, Aspect Described Hardware-Description-Language.

Each language is targeted for a subset of problems; hence, they are useful for certain tasks. In this case, our interests are in the signal and image processing domain. MATLAB is an excellent tool and a language for simulation of signal and image processing solutions that has to be executed on a PC platform as outlined earlier in the introduction. The main motivation came from the study of image and signal processing problems that could not be solved on a typical computing environment; therefore, FPGAs are recommended, but it is hard to convert the algorithms and ideas into the programming due to the implementation issues.

The technology is developing as per Moore's Law [1], the PC might be up to the tasks and FPGAs are no longer required; however, the FPGAs could still be used for more complex tasks as people become interested in solving more complex problems. Some of the uses of FPGAs in these applications have been shown in Chapter 1.1.

1.4 Scope and the Goal

We intend to use the state-of-the-art technologies, including aspect-oriented features in software programming languages, and provide some MATLAB concepts directly to the hardware programming languages.

The aim of this project is to provide a high level programming language that is easy to program and also to produce actual working hardware devices. The benefits of high-level languages are considerable and include the following:

- Earlier time to market through faster implementation
- Higher programmer productivity
- The intent and readability of the code is more apparent
- Easier maintainability of the code

[1] Moore's Law states that the number of transistors on a chip doubles about every two years [58].

- Easier to experiment with concepts and coding methods
- Closer to mathematical and natural language expression

In order to meet the requirements and produce a compiler, the followings are the key issues to be researched:

- Design of the language definition using BNF
- Investigation of the available tools for compiler writing
- Research into AOP (Aspect-oriented programming)
- Implementation of optimizing compiler
- Programming in HDL (Hardware description language)

These are very broad research areas; therefore, the scope of this thesis will be limited to the basic functionality of the working compiler, implementation of the aspect features and back-end generations. Even though we have implemented an optimising compiler, the underlying theory and techniques in optimisation are not within the scope of this thesis.

At this stage, the compiler produces synthesizable VHDL language as an output. The generated VHDL language can be re-compiled using a VHDL compiler to generate the final netlist. In the future, this intermediate stage will be bypassed and our compiler will generate the netlist directly. An overview of this project is given in Figure 1.3.

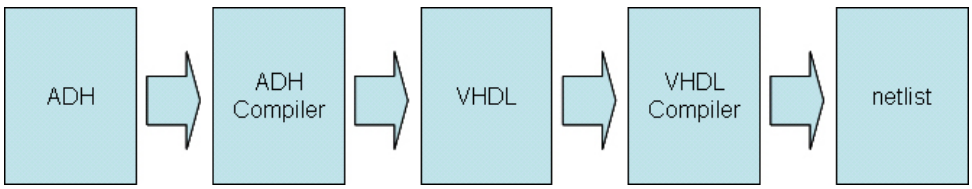


Figure 1.3: Overview of ADH and its synthesis mechanism

1.5 Typographical Conventions

We have also used the following typographical rules to make reading easier:

'Single quote' is used for a variable or an emphasis is required to distinguish the terms, e.g., variable 'a', 'state machine'.

Italic font is used for keywords or the words that might get confused with English words, e.g., *Boolean*, *aspect*.

1.6 Outline

This thesis can be summarised as follows:

- The core of what the signal and image processing problems are and why FPGAs are used for such problems
- Current problems in programming FPGAs
- A new programming model based on the evolution of software programming languages for microprocessors
- The compiler writing process in general and the implementation of this particular compiler

We will briefly provide a road map of the chapters: The next chapter describes the background information required prior to development of this new language and compiler. It shows what FPGAs are, how they differ from microprocessors, and how they are typically programmed in VHDL. The VHDL language itself is the final result of what the compiler produces at this stage. Therefore, a section with a VHDL language is provided and an extensive tutorial is also provided in Appendix B. AOSD (Aspect-Oriented Software Development) is also discussed and for one of the aspect-oriented languages, *AspectJ*, its tutorial is provided in Appendix C.

Chapter 3 introduces the features of the new language, ADH. The focus of this new language is in aspect-oriented support; therefore, most pages are spent on the aspect-oriented features. Chapter 4 gives an overview of the compiler writing process in general. We discuss the front-end in more detail in this chapter.

Chapter 5 explains the implementation method of the front-end of the compiler and how *weaving* is implemented. Chapter 6 describes the intermediate-stage implementation of our compiler and it is discussed with examples in detail throughout this chapter. Chapter 7 is the back-end of the compiler; the details of code-generation from the intermediate-stage are described.

Chapter 8 shows the current results and analysis of what has been produced. This chapter also states some of the remaining problems (current bugs) that will need to be dealt with in the future. Chapter 9 gives the discussion and the conclusion with suggestions for future work.

1.7 Publications

Some components of this thesis have been published in the following conference papers:

- Su-Hyun Park and Andrew Bainbridge-Smith. ADH: Aspect Described Hardware-Description-Language. In *Proceedings of the twelveth Electronics New Zealand Conference, ENZCon05*, pages 69–74, 14–15 November 2005, Auckland, New Zealand. [60]

- Andrew Bainbridge-Smith and Su-Hyun Park. ADH: An Aspect Described Hardware Programming Language. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, FPT 2005*, pages 283–284, 11–14 December 2005, Singapore. IEEE 2005. [10]

Chapter 2

Background

This chapter is an overview of the underlying technologies related to this project. It includes an overview of FPGAs and programming them using VHDL. The VHDL language structure is briefly described and the disadvantages are outlined. Also presented is a discussion on the evolution of the software programming languages for microprocessors and software engineering in general, which leads to an aspect-oriented programming.

2.1 Field Programmable Gate Array

All computing devices consist of collections of logic gates ("AND", "OR", etc.) and memory cells (flip-flops) arranged to solve logic or arithmetic functions. For the microprocessors, these logic gates are in a predetermined arrangement, as are ASICs. PLDs (Programmable Logic Devices) are not in a predetermined arrangement and the logic functions can be programmed. A PLD contains arrays of "AND" and "OR" gates and they are programmed by blowing the fuses in the PLD to implement gate operations. An FPGA (Field Programmable Gate Array) is also a type of logic chip that can be programmed; therefore, they fall into the class of PLDs even though the technology used is different. An FPGA is similar to a PLD, but the FPGA is memory[1] based; hence, it contains logic lookup tables instead of "AND" and "OR"

[1]Based on SRAM (Static Random Access Memory)

gates that PLDs contain. PLDs are generally limited to hundreds of gates whereas FPGAs support thousands of gates. They are especially popular for prototyping integrated circuit designs. Because of their parallelism and great savings in power consumption, they are very effective in solving some signal and image processing problems as well, compare to multi-microprocessor solutions.

Figure 2.1 shows conceptual construction for an FPGA: logic blocks in a sea of interconnecting buses. The typical FPGA logic block consists of a four input lookup table (LUT) and one output. The logic lookup-table and interconnecting routing channels are configurable as shown in Figure 2.2. Whenever the vertical and horizontal wire channels intersect, there is a switch box that allows it to connect to three other wires in adjacent channel segments.

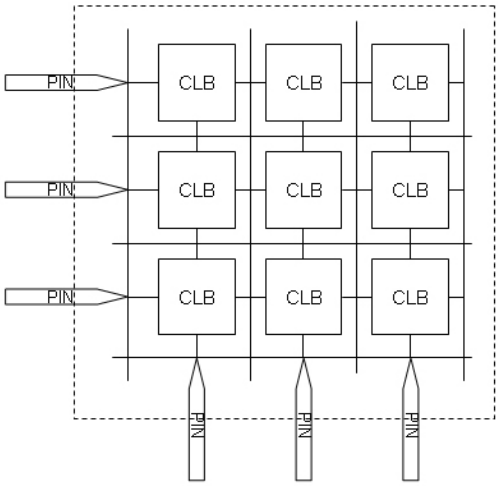


Figure 2.1: Conceptual construction of an FPGA with configurable logic blocks (CLBs) in a sea of interconnecting buses

The configuration that defines the behaviour of the FPGA is typically done using schematic diagrams or HDL languages. The tools, usually provided with the chip vender, produce netlists from the schematics or HDL languages, which are mapped in the hardware using a technique called place-and-route. They are then verified by a developer through simulation and timing analysis. Once the design and validation process completes, the binary file is generated (again, using the proprietary software provided by chip vender) that is used to configure or reconfigure the FPGA device; see Figure 2.3.

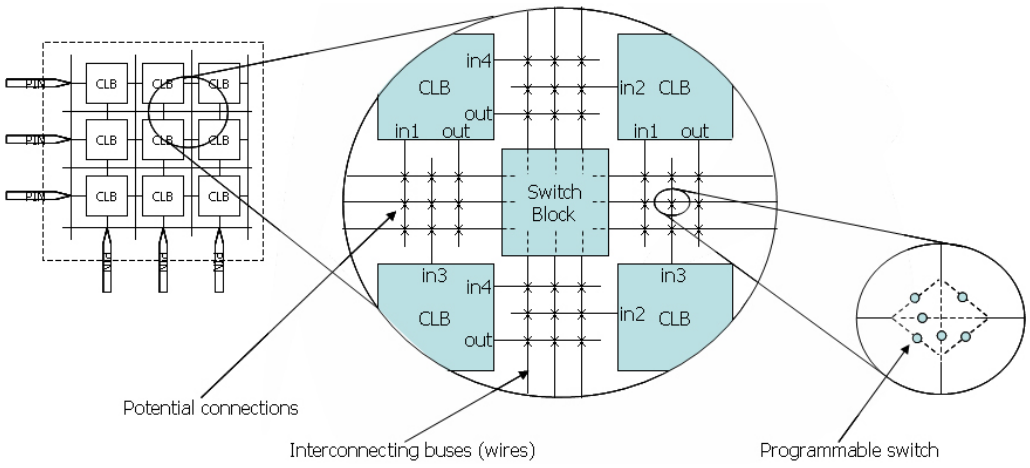


Figure 2.2: An FPGA with CLBs and the routing channels in more detail

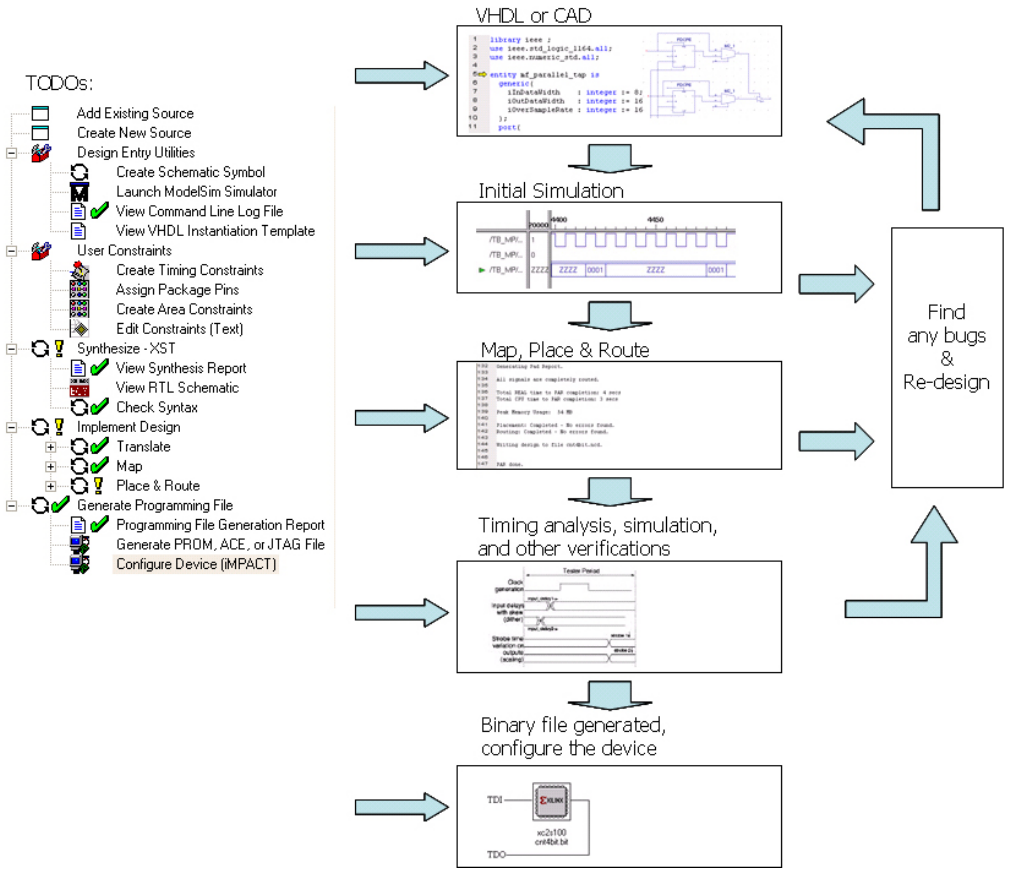


Figure 2.3: Overall FPGA configuration process

The architecture and design of FPGAs are shown in this section and the following section is a discussion on the programming of these devices.

2.2 Programming languages for hardwares

The first development of reconfigurable hardware was started from schematic diagrams with the help of CAD (Computer Aided Design) tools. It was a nice idea for educational purposes and was effective for small size circuits; however, designing larger size circuits with CAD tools was not possible. The problems in schematic diagrams are as follows:

- **The maintenance of the schematics can be difficult because the intent of the design is often hidden by its implementation.** (The abstraction problem)
- **A schematic must often be accompanied by documentation to describe a designer's intent and the functionality of what the schematic is representing. However, the descriptions in natural languages can be interpreted in many different ways.**
- **Schematic captures are difficult to re-use, so a designer who works in a schematic capture for one project may not be able to re-use material when working on a new project that requires the use of a new schematic capture environment.**

Hence, the programming idea arose. There are now two major industry standard HDLs, VHDL and Verilog. Both of them were developed in the mid 1980s and are similar in structure; therefore, we only discuss VHDL in this section. An extensive VHDL tutorial is also provided in Appendix B on page 117.

2.2.1 What is VHDL?

VHDL stands for VHSIC (Very High Speed Integrated Circuits) HDL (Hardware Description Language). It was first developed by the U.S. Department of Defence

and sponsored by the IEEE in the mid–1980s. The VHDL was standardised by the IEEE by 1987 and known as VHDL–87. This language has been revised and a new version, VHDL–93, has been released and has been widely adopted now [18, 61].

2.2.2 Structure of the language

VHDL has two parts known as *entity* and *architecture*. The *entity* defines the input and output ports, in other words, it defines the interface or prototype. The *architecture* defines how the output is generated from the inputs. There are two distinct method writing the *architecture* that is explained in the following section.

The statements written in *architecture* are executed concurrently. Even though the statements are written in sequential order, one line after another, they all have the same precedence and are all executed at the same time.

2.2.3 Structure of the model

VHDL can be modelled by two distinct methods known as 'behavioural' and 'structural' methods. The behavioural method describes how the outputs are generated by the inputs. It is done by the assignment of the I/O variables. It describes the dataflow within the hardware. The structural method gives the interconnection between the blocks. Engineers who have a lot of experience in software programming will tend to program in the behavioural model whereas hardware engineers, who like thinking in terms of gates, will tend to program in the structural model. Each of them has advantages and disadvantages that we will briefly go over. In real life, a combination of both methods will be used.

The behavioural method gives more abstraction and hence more high-level programming and the structural method gives lower-level programming. The same logic can be implemented in either the behavioural or structural models. It is usually good to combine the two methods together where appropriate. For example, a full adder can be implemented by a behavioural method and a four-bit adder can be implemented by the structural method by implementing a full adder.

The behavioural method can contain one or more *process* blocks. We have men-

tioned earlier that the VHDL statements are executed concurrently; however, the statements within a *process* are executed sequentially. The detail on how to program in the behavioural method, the structural method and the process are shown in Appendix B.

The difference of the two models is shown in Figure 2.4. Note that this does not mean that the structural methods are drawn with gate symbols. It gives a general idea of how to implement in VHDL in concept.

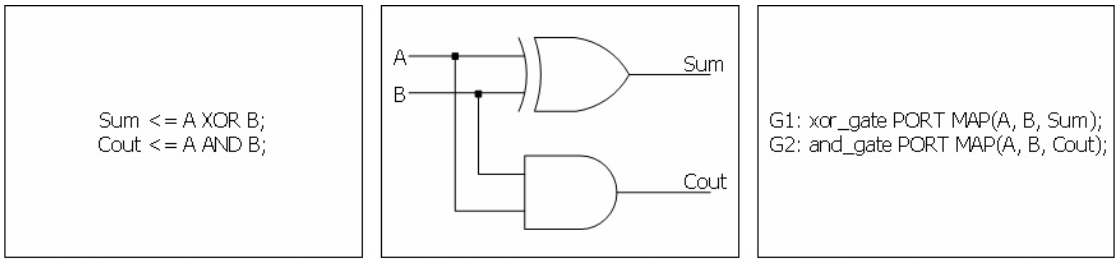


Figure 2.4: Different types of VHDL structure

2.2.4 Why not VHDL?

The programming languages developed in the 1980s did not solve all of the problems arising from the CAD method, for example, the abstraction problem (in Section 2.2) explained earlier. The maintenance problem that the old CAD method had still remains and more importantly, the intentions of the tasks are often hidden by its implementation.

VHDL is strong-typed language, which means the data types are not convertible whereas in *C* you can assign different data types to other types of variable. Therefore, a developer is required to code precisely defined and matching data types.

VHDL is case-insensitive, which can cause some difficulties in identifying the variables while debugging, and it is a good software idea to write all the keywords in capitals.

In VHDL, every statement is executed concurrently unless written in a *process*. The concurrent execution is the same for most hardware programming languages, but for the programming languages for microprocessors, the codes are executed

sequentially, one line after another. However, the programming language must be written in a sequential way and since this is how humans read the codes as well, it causes novice users difficulty in learning and conceptualising VHDL.

The structure of VHDL can be written in two distinct ways, structural and behavioural, which makes it inconsistent and makes it difficult to understand in some cases. And for similar tasks, there are different types that has to be used, for example, the use of *signal* and *variable* can be confusing.

The same statements declared in an *entity* must be re-defined when used as a *component* or used in a *package*. These statements are verbose and repetitive, making it tedious and error-prone for the developers.

VHDL is known as a complex language. Since it is designed for general purposes, the codes are not specific and become longer, hence more error-prone. Most of all, the main problem is the conversion of a task specific algorithm into VHDL is very difficult due to implementation issues, the lack of high-level design methodology, and difficulty in design reusability.

The listing in Figure 2.5 shows an example of a 4-bit adder using *component* and written in the 'structural' method. In the listing, 'PORT MAP' at 'stage3' uses a different writing style compared to the other stages. For larger adders, say an 8-bit adder, the different stages can even be implemented using a *for* loop and generate eight statements. There are many different ways of expressing the same tasks. It makes the program hard to debug and makes it hard to track down the bugs when errors occur, which can be a major problem in development.

The three listings in Figure 2.6 show a comparison between VHDL and high-level software programming languages, *C* and *Java*. The functionality of the listings are printing out "Hello world", which we all come across in first time programming. Clearly, the abstract idea is simple — print out two words — therefore it would be nice to have a similar high-level programming language in hardware that could simply be done in software programming. The following section is a discussion of the evolution of software programming languages.

```
 1:  LIBRARY ieee;
 2:  USE ieee.std_logic_1164.all;
 3:
 4:  ENTITY adder4 IS
 5:    PORT (Cin            : IN  STD_LOGIC;
 6:          x3, x2, x1, x0 : IN  STD_LOGIC;
 7:          y3, y2, y1, y0 : IN  STD_LOGIC;
 8:          s3, s2, s1, s0 : OUT STD_LOGIC;
 9:          Cout           : OUT STD_LOGIC);
10:  END adder4;
11:
12:  ARCHITECTURE Structure OF adder4 IS
13:    SIGNAL c1, c2, c3 : STD_LOGIC;
14:    COMPONENT fulladd
15:      PORT (Cin, x, y : IN  STD_LOGIC;
16:            s, Cout   : OUT STD_LOGIC);
17:    END COMPONENT;
18:  BEGIN
19:    stage0: fulladd PORT MAP(Cin, x0, y0, s0, c1);
20:    stage1: fulladd PORT MAP(C1, x1, y1, s1, c2);
21:    stage2: fulladd PORT MAP(c2, x2, y2, s2, c3);
22:    stage3: fulladd PORT MAP(Cin=>c3, Cout=>Cout, x=>x3, y=>y3, s=>s3);
23:  END Structure;
```

Figure 2.5: A 4-bit adder in VHDL

```
1:  --  VHDL example programme: hello.vhd
2:  use std.textio.all;
3:
4:  entity hello is
5:  end entity hello;
6:
7:  architecture Behav of hello is
8:      constant message : string := "Hello world";
9:  begin
10:     process is
11:         variable L: line;
12:     begin
13:         write(L, message);
14:         writeline(output, L);
15:         wait;
16:     end process;
17: end architecture Behav;
```

```
1:  % C example programme: hello.c
2:  #include <stdio.h>
3:
4:  void main() {
5:     char message[] = "Hello world";
6:     printf(message);
7:  }
```

```
1:  // Java example programme: Hello.java
2:  import java.io.*;
3:
4:  class Hello {
5:     public static void main(String[] args) {
6:        String message = "Hello world";
7:        System.out.println(message);
8:     }
9:  }
```

Figure 2.6: “Hello world” listings in *VHDL*, *C* and *Java*

2.3 Programming languages for microprocessors

Programming languages for microprocessors, in other words, software programming languages, are always evolving. They have evolved since personal computers have been widely adopted and more people are developing applications using their favourite programming languages. The reason for evolution is because, as our experience grows (perhaps in a particular application domain), we find there are (or could be) better ways of doing things.

The first programming languages were *Assemblers*, which we still use in some applications. They are tied to the architecture of the machines; therefore, they were more of machine-friendly languages. One program written to operate on a particular machine cannot be operated on other machines that have a different architecture. Hence, it was necessary to develop new languages that could more easily be ported to other machines, this is done by compilers. It was also advised to modularise a program into smaller parts (functions or procedures), so they can be used in other parts of programming or as a division of labour. These new types of languages are called functional and procedural languages and fall into high-level programming languages as they are human-friendly languages. A well known programming language, *C*, is a high-level procedural language that also falls into the category of imperative languages. It is still a very widely used language but there were some difficulties when many people were working on the same project sharing their parts in the programming. With imperative languages, the division of a program is typically done by the top-down (or bottom-up) approach, which breaks up the program into functions that describe the way of doing things. This type of breaking up of the codes does not work well in software engineering [21], which lead to the OO (Object-Oriented) design and people starting to look at a program in different ways.

The imperative languages still involve thinking in computer nature[2], which is not exactly how humans think. From the basis of the imperative languages, object-oriented languages have developed. It was more like how a human thinks (communication between the objects) and each person developing each object is an easier

[2]Thinking of data-flow and control-flow. For example, the use of pointers in *C*

development process in a big project. The idea behind OOP (Object-Oriented Programming) is that a computer program may be seen as composed of a collection of individual units (or objects) that act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions or procedures. In traditional programming, the functions and procedures call each other, and are simply a list of instructions to the computer but in OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. This made development process easier, gives more flexibility, and allows a large scale development environment.

Table 2.1 summarizes the evolution of software programming languages.

Generation	Year	Languages and main features
1	1940s	Machine code
2	1950s	Assembler
		– no structure & elementary control (test & goto)
3	1960s	Structured programming (*FORTRAN, COBOL, C*)
		– basic control (if then else, while, for)
		– modular programming (functional and procedural)
4	1980s	Object-oriented programming (*Java, Smalltalk*)
		– more sophisticated modular programming
	∼1995	Aspect-oriented programming (*AspectJ*)
		– better ways of partitioning a problem

Table 2.1: Evolution of the programming languages for microprocessors

Why object-oriented programming?

OOP is often called a paradigm rather than a style or type of programming to emphasize the point that OOP can change the way software is developed by changing the way that programmers and software engineers think about software. Following is an example illustrating OO design.

There are often several data structures (objects) that have to be treated similarly in some respects, but differently in others. An example is drawing pictures: circles,

triangles, stars, and others. We want to apply some actions to some of them, for example, 'draw' to display on screen. However, 'draw' is different for every object. In traditional programming, we could implement 'draw' as in a big 'SWITCH CASE' control structure. This statement needs to be modified as well as the shape itself if another shape is added. However, in OOP thinking, each of the pictures fall into a 'shape' category and the picture objects would tell a 'shape' object, "Here's a construction of this shape; you figure out how to display on screen.".

OOP gives more flexibility, easier maintenance, and is easier to learn for those new to computer programming than previous approaches. The translation from real-world phenomena to objects (and vice versa) is easy because there is direct mapping from the real-world to the object-oriented program. The OO design paradigm and implementation in OOP can be found in many journal articles [22, 54, 59].

OOP is a natural progression from functional and procedural programming that has seen particularly strong uptake in programming of graphics. The HDLs can also be represented in graphical ways such as an FSM (Finite State Machine) or schematic diagrams. Therefore, we should expect an OO langauge to be well suited to this class of problems.

2.3.1 Aspect-Oriented Programming

OOP was not the end of the evolution. People were still not happy about some of the things in OOP. An object consists of data and associated *methods* that process the data. But some data components could be involved in two or more objects, whereas some data components do not naturally belong to any object; therefore, the programmer had to choose where to put those data that are not trivial. Many objects may also have similar functionalities written into them. The codes are not identical; therefore, they cannot become a common function but they do a similar role in parts of programming. These motivated a new language paradigm called AOP (Aspect-Oriented Programming) also called AOSD (Aspect-Oriented Software Development) [17,25]. It is not a totally different way of programming. It is built on top of OOP. The main idea behind OOP is breaking down of a program into objects, and the main idea behind AOP is the breaking down of a program into distinct parts

where there is minimum overlapping of the functionality. In AOP, some of the data and functions that we have just talked about can easily be resolved. A new object called *aspect* contains the data and *methods* that can be spread across many different objects.

AOSD is a new paradigm that attempts to aid the programmers in the separation of concerns. A concern is any piece of software focused on one thing. In OOP, this concern can be separated into the *classes* and *methods*. AOSD is used to solve some of the problems that the conventional object-oriented concepts have. One of the problems is the partitioning problem, shown in Figure 2.7. In OOP, as stated earlier, each object has its *attributes* for storing data that belong to an object, and the *methods* that modify the value of those *attributes*. However, the division of those *attributes* and *methods* is not trivial, as shown in Figure 2.7. We may wish to share variables or functionality among multiple objects. The solution in OOP yields the split of a *method* into multiple objects, hence, breaking good encapsulation and make high-coupling, as shown in Figure 2.8. It can be solved using the AOP approach. Figure 2.8(a) shows the splitting of the *method* when one way is chosen in the object-oriented model and Figure 2.8(b) shows grouping the spread functionality back using the aspect-oriented model. Hence, good encapsulation and low coupling is maintained in AOP. There is an alternative solution using *inheritance* in OOP. This might create unnecessary hierarchies of objects and might be difficult to work with. AOP simply extends this concept.

2.4 Summary

Table 2.2 is an overview of how the software programming languages and hardware programming languages have evolved. The introduction of each higher level programming paradigm effected an enormous productivity improvement through greater abstraction of the problem. Designers are able to focus on the behaviour and essence of the problem rather than on its implementation. In other words, the designers need to do little work translating the conceptual design into the implementation design. In contrast to the software programming languages, hardware programming

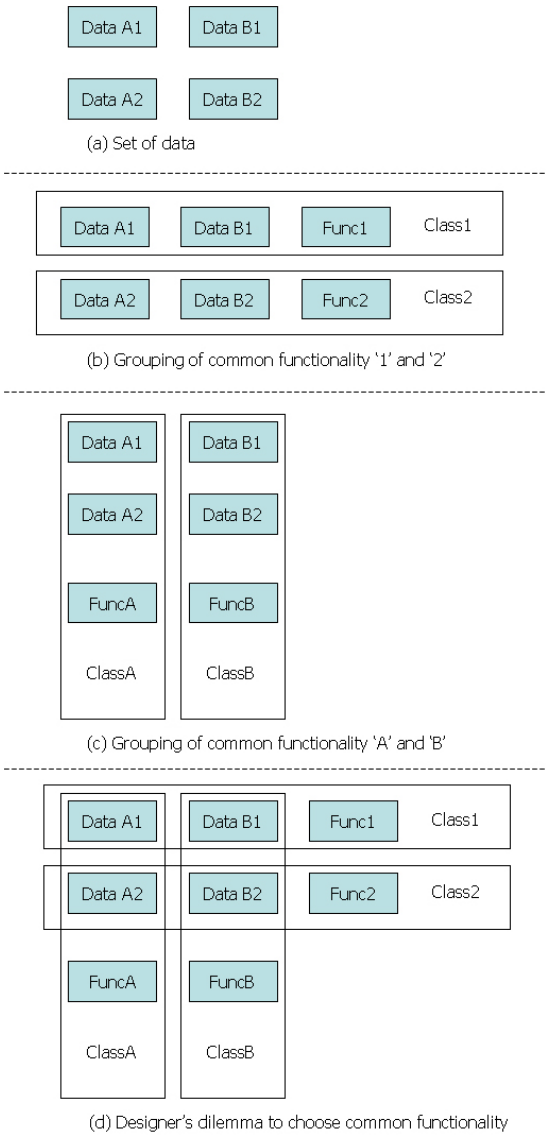


Figure 2.7: Non-trivial partitioning problem

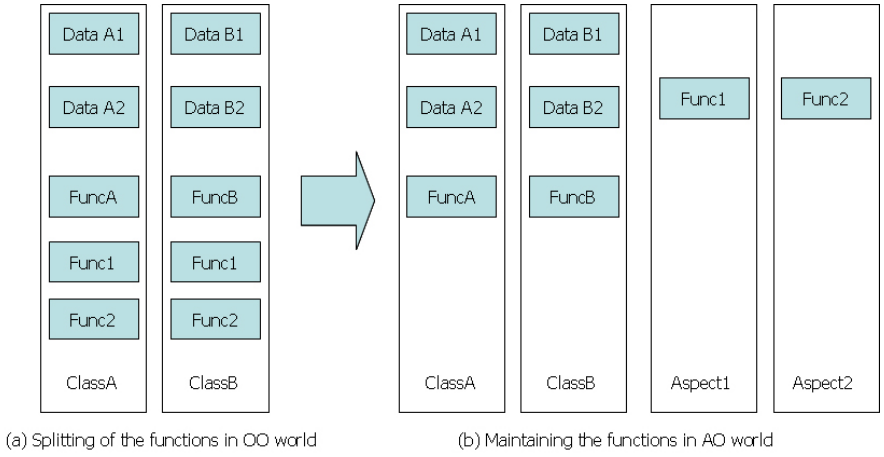


Figure 2.8: Solution to the non-trivial partitioning problem using OOP and AOP

Software programming languages	Level of abstraction	Hardware programming languages
	High	
	Human language	
	Mathematical expression	
AspectJ, AspectC etc.	Aspect-Oriented	**ADH**
Java, C#, C++	Object-Oriented	
C, Pascal, Fortran, Cobol	Functional & Procedural	VHDL, Verilog
Assembler	Assembler	Schematic, Gate Logic
	Machine code	Transistor level
	Low	

Table 2.2: Evolution of software and hardware programming languages

language development is not up-to-date. VHDL and Verilog still use functional and procedural based language structures and they require a lot of thought in digital logic implementation rather than in the application and its algorithm.

In this chapter, we have described the evolution of programming languages and why it is necessary to have a higher level programming language for hardware programming. The next chapter describes our new proposed language, ADH.

Chapter 3

Aspect Described Hardware-Description-Language

In this chapter, we outline a new language called ADH. This language is targeted at the control modelling, and the signal and image processing domains, hence the domain specific features such as vector and matrix operation are built-in to the language. This is useful in converting the ideas and simulations from sources such as MATLAB easily onto hardware devices. The *aspect* features are modelled from one of the well-known aspect-oriented programming languages, *AspectJ* [2]. In this chapter, many syntactic features are described with a particular emphasis on the aspect-oriented support.

3.1 Aspect-oriented support

Aspect-oriented programming extends the concept of object-oriented programming to allow the physical, as in written, separation of code from the point of execution [25]. While naively this may be thought to make code harder to read, it has the opposite effect. Syntactically, an *aspect* looks just like a *class* but it is not instantiated through the use of a *new*. Instead it provides two major mechanisms for interacting with *classes* — static cross-cutting and dynamic cross-cutting.

Take a look at the following example first.

```
class AlgorithmDesigner {
   a <- b + c;
   x <- y;
}
```

Imagine this is a complete solution from the signal and image processing engineer's view. However, when implementing this onto a hardware device, it is incomplete. Consider the expression of '`b + c`'; the abstract concept of this expression should be readily apparent to the reader, but its implementation is not so straightforward. One has a whole host of potential adder routines available, ranging from the sequential full adder through to carry-propagate adder designs [52]. Moreover, one must also consider such issues as number representation and bit width. The number of bits set at one point in the processing need not always be the same for all points; a different number of bits can be used at different stages.

Taking all the hardware engineer's concerns into account and writing a code that will satisfy them will decrease the readability of the code, reduce the abstraction that the code had before and make it difficult to share the source code. Therefore, we contend that aspects are a natural occurrence of abstracting programs for implementation on FPGAs (and digital logic in general) and demonstrate this through the following seemingly simple problem, which is an abstract illustration (similar to the pseudo-code style) of how aspect could be used for the problem described earlier.

```
aspect HWdesigner {
   capture : (a <- b + c;) {
      a <- carryPropAdder(b, c);
   }
   capture : (x <- y;) {
      x <- (Integer4bit) y;
   }
}
```

Finally, it is possible to have many different adders used throughout a design. Nevertheless, it is desirable to write this abstract level of expression and to provide elsewhere instructions on specific implementation, thus separating our concerns or aspects. This separation of concerns allows application and hardware programmers to focus on their separate issues.

3.1.1 Cross-cutting elements

The aspect features are added into ADH language with the weaving rules set by the following concepts and commands — *join point*, *pointcut*, *advice* and *introduction*.

Join point

A *join point* is an identifiable point in the execution of a program. It specifies points within the execution control graph of a program where advice can modify its behaviour. For example, a *method* call is a common *join point*, as is the assignment of a *class* variable as shown in Figure 3.1. Recent works have also shown how this technique can also be extended to loop structures, such as the *for* loop [40].

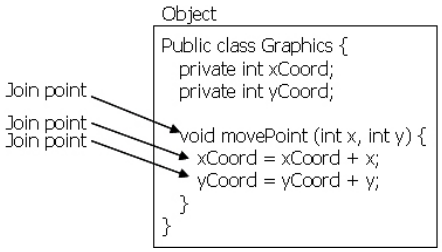


Figure 3.1: Example of *join points* on an object

Pointcut

A *pointcut* is a program construct that selects *join points* and collects context at those points. It addresses the location to be intercepted in the control-flow graph and does not change any behaviour until *advice* is given.

The following is an example of a *pointcut* that captures at the call of the 'sayIt()' *method* that exists in the 'Hello' *class*.

```
Pointcut captureHello() : call(method() = Hello.sayIt());
```

Advice

An *advice* is the code to be executed at a *join point* that has been selected by a *pointcut*. It specifies what behaviour to add into the execution graph. The *advice* can add behaviour before, after or around (or instead of) the operation specified at the *pointcut*.

The following is an example of an advice adding some actions before calling the 'sayIt()' *method* that is captured by the previous 'captureHello()' example.

```
before() : captureHello() {
   // do some actions.
}
```

One of the aspect-oriented languages we have modelled, *AspectJ*, requires the use of additional commands — e.g. *args*; see Appendix C for more detail — to capture the arguments of the *method*. However, ADH automatically captures the arguments of the captured *method*. So both the input and output arguments have the scope to be used in the capturing *advice*.

Introduction

An *introduction* is a static crosscutting instruction that introduces changes to the *classes*. It directly adds *attributes* and *methods* to an existing *class*. The following is an example that adds a variable and a *method* to an existing *class*, 'Hello'.

```
String Hello.newMessage = "Hello, World."

method (Integer sum;) = Hello.addTwo(Integer a; Integer b;) {
   sum <- a + b;
}
```

Aspect

These techniques allow previously written *classes* to take on additional behaviours, as shown in Figure 3.2; i.e., more *attributes* and *methods* without directly editing the *class* files. This form of encapsulation captures common functionality within one spot, the *aspect* file, while making the behaviour available to a widely spread set of *classes*. The following is an example of an *aspect* that can contain *introductions*, *pointcuts* and *advices* declarations.

```
aspect Example1 {
   // introductions
   // pointcuts
   // advices
}
```

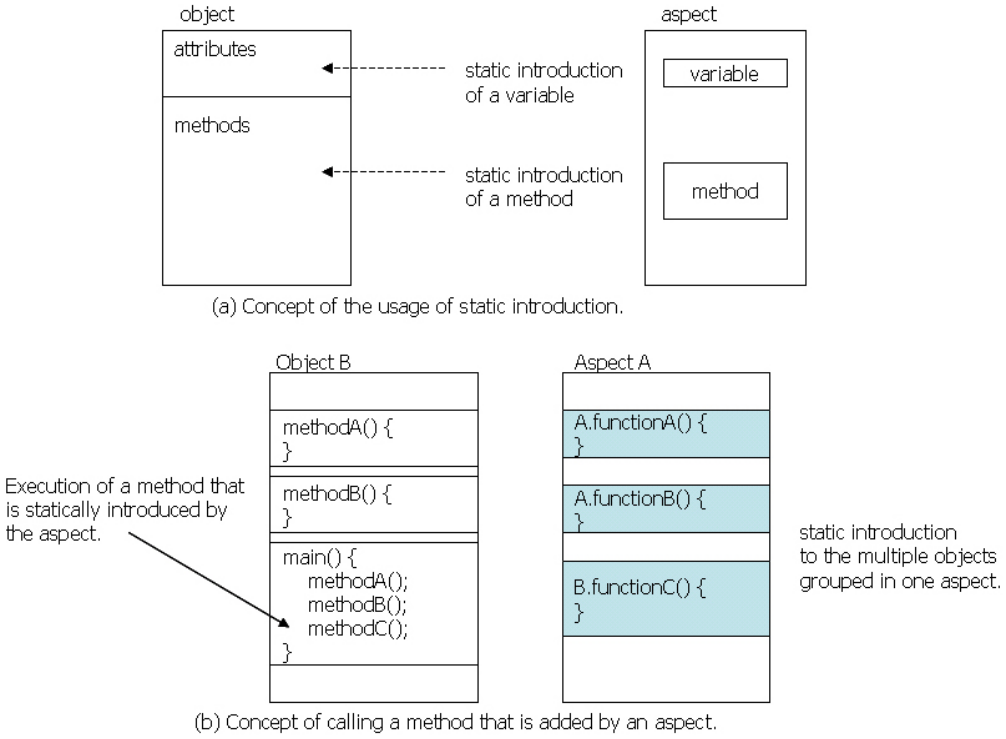


Figure 3.2: Aspect usage examples

3.2 Weaving

The *aspects* are the high-level concepts that the developers specify as the functional and non-functional requirements. In team work, the developers could be split into multiple teams and focus on the separate concerns that are written as *aspects*. Hence, these aspect-oriented concepts somehow need to map into the structured programming style. The compiler uses a process known as *weaving* to spread the aspect into multiple objects.

Figure 3.3 shows how the aspect can be used to keep the separation of the concerns, and how weaving spreads the codes back to multiple objects. The system core can concentrate on its own tasks, such as an algorithm for a digital camera, then the house-keeping functions such as debugging and optimisation can be grouped in their own aspect modules. The *weaving* process spreads these functionalities out to the complete system.

In order to implement this *weaving*, there are two different cross-cutting styles that need to be looked at — static cross-cutting and dynamic cross-cutting.

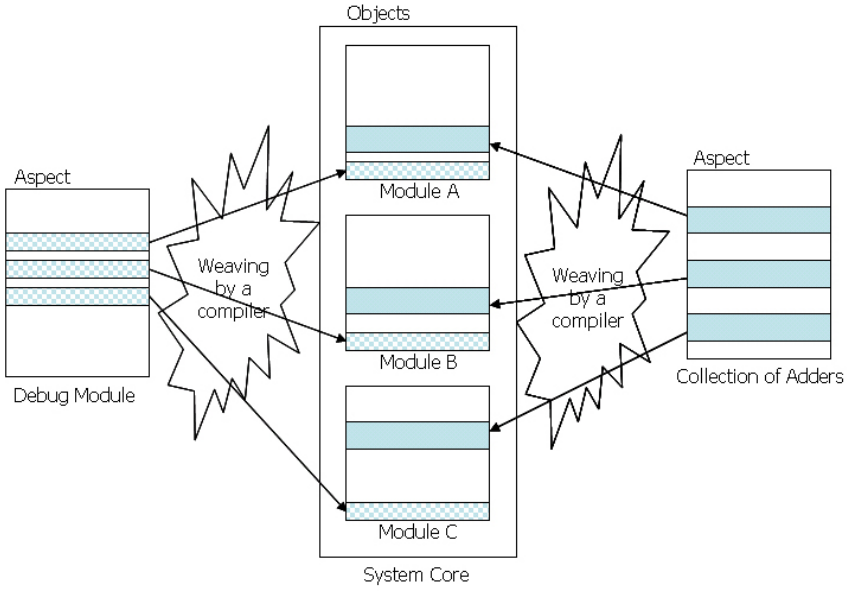


Figure 3.3: Separation of the concerns using *aspects* and the *weaving* to form a complete system

3.2.1 Static cross-cutting

Static crosscutting is a technique that allows a change in behaviour at compile-time. It is done using forms called *introduction*. The *introductions* are defined in an *aspect* but they modify members of other *classes*. With the *introduction*, we can add *attributes* and *methods* to existing *classes*.

Introduction is a powerful mechanism for capturing crosscutting concerns because it not only changes the behaviour of components in an application but also changes their relationships.

3.2.2 Dynamic cross-cutting

Dynamic cross-cutting is the weaving of new behaviours into the execution of a program. It is able to capture the arguments of the modules or cut across the modules to modify the system behaviour. Certain execution actions can be bypassed or replaced. You can even specify the weaving points and the action to take upon reaching those points in a separate module. These are done using forms of two cross-cutting elements — *pointcut* and *advice*.

Some of the dynamic cross-cutting can be determined at compile-time and can

be processed in a static behaviour. Therefore, the dynamic cross-cutting can be split into the compile-time cross-cutting elements and the run-time cross-cutting elements. For this reason, we will be treating all the compile-time cross-cutting as a static cross-cutting (strictly speaking, they still fall into dynamic cross-cuttings).

Figure 3.4 illustrates the difference in the control flow between static and dynamic cross-cutting. The static cross-cutting is being executed "inline", while the dynamic cross-cutting is being executed as "calls".

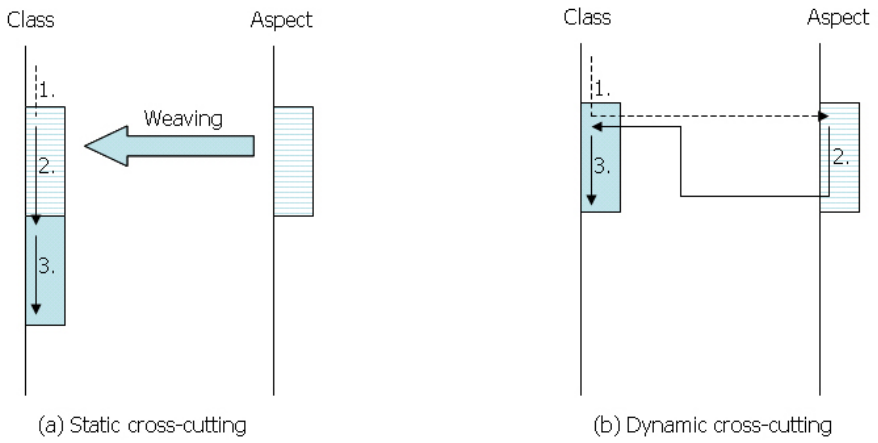


Figure 3.4: An illustration of the control flows of different *weaving* techniques (1. Reaching the *join point*, 2. Processing of an *advice*, 3. Back to the processing of a *method*)

3.2.3 Weaving techniques

For *AspectJ*, there are a number of papers presenting *weaving* techniques in [33, 38, 41], and clearer cross-cutting mechanisms in [6, 36, 42]. The *AspectJ* uses code-tree manipulation that occurs at the *Java byte-code*, which is the intermediate code[1] for *Java*.

In programming languages for microprocessors, the dynamic cross-cutting is more often used. Dynamic cross-cutting provides the control structure to alter execution at run-time; i.e., before the control sequence is fully known. This has to be woven at the intermediate-stage. However, our compiler uses a preprocessor

[1] *Java byte-code* is sometimes considered to be a back-end final code that runs on *JavaVM*.

Type	Description	Tool used
Compile-time	The source code from the primary and aspect languages is weaved before being put through the phases of the compiler where byte code is produced. AspectJ 1.0.x uses this form of weaving.	Compiler
Link-time	The weaving occurs after the primary and aspect language code has been compiled into byte code. AspectJ 1.1.x uses this form of weaving.	Linker
Load-time	The weaving occurs when classes are loaded by the classloader. Ultimately, the weaving is at the byte-code level.	Classloader under Java
Run-time	The virtual machine is responsible for detecting join points and loading and execution aspects.	Virtual machine

Table 3.1: AOP Weaving Types [31]

style 'code weaving' at the front-end; this is more fully explained in Chapter 5.2 on page 59.

The *weaving* can be done at four separate stages, as shown in Table 3.1. [2] In this thesis, we focus on the compile-time *weaving* and the static cross-cutting elements that are implemented by this process. These elements are the *introduction* and the compile-time dynamic cross-cutting elements (which are being treated as the static cross-cutting).

[2]Table adopted from the book *Mastering AspectJ* by Gradecki [31]

3.3 Control structures

The language provides the normally expected loop and control structures, although these carry some changes and restrictions to what would normally be expected on a microprocessor. For example, the *if, then, else* is written as *when, then, otherwise* to better capture the temporal meaning of the statement. Furthermore, all loops must eventually be run-time determinant and of finite length, so they can be completely unwound.

We also introduce a new control structure for explicitly writing finite state machine expressions, together with a *set* type for enumerations and state names. The *set* is akin to the *enum* provided in the latest specification of the *Java* language [48], and its implementation is more in keeping with that of a *class*, thus providing syntax consistency. This does provide great clarity and efficiency, as the majority of digital logic expressions can concisely be expressed in an FSM form. Figure 3.5 is an example usage of FSM.

```
 1:  public set VendingState {
 2:    READY; COIN_IN; GIVE_COKE; GIVE_CHANGE;
 3:  }
 4:
 5:  public class VendingMachine {
 6:     VendingState stateVar;
 7:
 8:     state(stateVar, clk) {
 9:        READY -> COIN_IN when (money == 10);
10:        READY -> COIN_IN when (money == 20);
11:        READY -> COIN_IN when (money == 50);
12:        COIN_IN -> GIVE_COKE when (total >= 100);
13:        GIVE_COKE -> GIVE_CHANGE when (total > 0);
14:        GIVE_COKE -> READY when (total == 0);
15:        GIVE_CHANGE -> READY when (total == 0);
16:     }
17:
18:     when (stateVar == COIN_IN) {
19:        total <- total + money;
20:     }
21:  }
```

Figure 3.5: FSM example using '*state*', '*when*' and '*set*'

3.4 Assignment and buffering

In order to reinforce that the programmer in ADH is written on concurrent software we have deliberately chosen the symbol '$<-$' for use in assignment. As indicated, each individual statement encompasses the parallelism concept. A run of statements is to be considered as sequential in nature; they will however be implemented as a set of concurrent statements provided they do not form a feedback path (dependency). Where a feedback path is identified, the path is automatically broken by an implicit buffer. Similarly, *if* statements without catching *otherwise* also implement implicit buffering. Consequently, all functions must be provided with a clock signal; again this is done implicitly via a master clock; however, the programmer does have control to override this default behaviour.

An example code of assignments that do not require any buffering and can be executed in parallel is as follows:

```
a <- b + 5;
c <- 2 * b;
```

And an example of assignments that has a feedback path and automatically broken by implicit buffers by the compiler is as follows:

```
a <- b + 5;   // Line 1
c <- c + a;   // Line 2
a <- 10;      // Line 3
```

In line 2, the statement is suspended until the variable 'a' is set by the previous statement, line 1. In line 3, it has to be suspended until the previous value of 'a' has been used by 'c' in the previous statement, line 2, or the previous value of 'a' is buffered that will be used by line 2 and the value of 'a' can be altered at line 3. (In fact, line 3 does not form a dependency with line 2 after variable renaming occurs by static single assignment (SSA) form; see Chapter 4.2.1 for SSA form.)

3.5 Standard operators

Table 3.2 lists the standard infix operators. Note in particular the offering, as in MATLAB, of native operators or Matrix/Vector objects. A mechanism for overload

and type checking the infix operators is provided, as is the ability to change their meaning and implementation through *aspects*.

Logical	`| (or), # (xor), & (and), ! (not)`
Comparative	`<, <=, >, >=, ==, !=`
Arithmetic	`+, -, *, /, %, ^ (power of)`
Vector	`.+, .-, .*, ./, .%,`
	`.^ (point wise operations), *, /,`
	`\ (matrix operations), @ (convolve)`
Assignment	`<-`

Table 3.2: The infix operators take on the usual mathematical order of precedence

3.6 Data Types

A number of data types and supported features for the data types are described in this section. Scalar data type and vector and matrices support are the main features and they are used through the use of *inheritance*.

3.6.1 Scalar data type

The target machine only operates at a single logic bit and can only perform logic gate operations, such as AND, OR and XOR. Therefore the language has only one 'built-in' data type, namely *Logic*. This data type is distinct from *Boolean* (which is a subset) in that a logic bit can have other states than *true* and *false*, such as *tri-state*, which provide important electrical characteristics that the programmer may need to control as defined by IEEE Standard 1164 [14]. The usual suspects of data types *Boolean, Integer, Real* and *String* that a programmer would normally expect to see are provided through a mixed view of built-in and library support. The mixed view is important because an integer has native support on a microprocessor with a certain fixed data width; whereas, on an FPGA, it does not. This has its advantages, as we can control and vary the data width of an integer at various stages, which

in turn provides more flexibility in controlling arithmetic error propagation and the SNR performance of an algorithm [19].

However, for many programming tasks this mixed viewpoint will appear transparent, providing seamless operation between two distinct code generation or execution phases of the compiler. The first runs on a target microprocessor allowing a user to input data, for example filter coefficients, will shape the final form of the generated hardware. This phase also allows information to be printed to the screen and other basic I/O facilities. In order for the programmer to express ideas elegantly, *Boolean, Integer* etc. are treated as internal native objects for this view. However, when used to express data objects on the target FPGA, they are provided with standard library support. Library functions provide for operations that convert the object into vectors of *Logic* bits, and overload facilities for infix operators.

All of these objects can be extended through the mechanism of *class* inheritance. Like *Java*, only provision for single inheritance with explicit interface specifications to express inter-class common behaviour is provided.

An example of scalar number types is given as follows:

```
class Natural {
  ...
}
class Integer extends Natural{
  ...
}
class Real extends Integer {
  ...
}
```

And an example of different length *Integers* that belongs to one family is as follows:

```
class Integer {
  ...
}
class Integer16 extends Integer {
  ...
}
class Integer32 extends Integer {
  ...
}
```

3.6.2 Vectors and matrices

The language also supports matrix and vector objects as first class objects. The power of MATLAB is in its central treatment of these objects and its supporting syntactical operations such a (*) for multiplication and (.*) for pointwise multiplication. This is a natural fit to the problem domain of signal and image processing and allowing programmers to efficiently implement solutions that are close to their natural mathematical abstraction.

It should be noted that the matrix/vector operations are inherently parallel operations and the matrix multiplier (*), for example, very succinctly captures this concept. Thus, when examining a *method* or function body, the programmer will have written a number of sequential steps, where each step encompasses a parallel concept, i.e. dataflow is parallel and control is sequential. This form of expression is very important, as it leads to superior optimisation and scheduling of the algorithm [13]. Interestingly, VHDL presents the programmer with the alternative viewpoint, a list of parallel statements that are sequential in nature (or parallel control and sequential dataflow). This viewpoint can be confusing to the novice programmer and makes optimisation and scheduling of the code more difficult for the compiler.

Consider the one-dimensional convolution of a signal $x(t)$ with a system $h(t)$. This can be expressed mathematically as, `y(t) = x(t)` $\odot$ `h(t)`, where $\odot$ is the convolution operator. Figures 3.6 show equivalent programmatic solutions in VHDL whereas ADH provides the convolution as built-in operator. The shear conciseness of the ADH solution not only better convoys the intent and abstract idea, this is not mere syntax sugar, but offers much greater scope for the compiler to interpret and optimise this expression (as well as the programmer through the use of *aspects*).

3.6.3 Automated type conversion and mapping

Matrix and vector datatypes are naturally common in hardware implementations, as eventually even scalar datatypes like *Integer* can be considered to be a vector of bits. However, the scalar datatypes of *Boolean, Unsigned, Integer* and *Real* are so widely used in natural descriptions of physical problems that we have provided automated

```
 1:  library IEEE;
 2:
 3:  entity Convole1D is
 4:    port (
 5:      image: in INT_ARRAY;
 6:      window: in INT_VECTOR;
 7:      output: out INT_ARRAY;
 8:      };
 9:  end Convole1D;
10:
11:  architecture Convolve1D_arch of Convole1D is
12:
13:  begin  -- Convolve1D_arch
14:    process
15:      variable i : integer;
16:      variable j : integer;
17:      variable k : integer;
18:      variable tmp_vector : INT_VECTOR;
19:      variable tmp : integer;
20:      begin
21:        for k in 0 to image_rows loop
22:          for i in offset to image_cols - offset loop
23:            tmp_vector := image(k, i-offset to i+offset) * window;
24:            for j in 0 to 2*offset+1 loop
25:              tmp := tmp + tmp_vector(j)
26:            end loop;  -- j
27:            output(k,i) <= tmp
28:          end loop;  -- i
29:        end loop;  -- k
30:  end
31:
32:  end Convolve1D_arch;
```

Figure 3.6: VHDL code segment to perform a one-dimensional window (convolution) on an image

type conversion. This is relatively straightforward for constant values, but does take on additional complexity for runtime computed terms. The default behaviour is to treat signed numbers as twos-complement and reals as fixed-point precision. In most situations, worst case bit lengths can be computed; although, there are better alternatives as outlined by other researchers [19, 26] and commented on in the summary.

Similarly, we automatically provide a form of type conversion when calling functions where there is a dimensionality mismatch. For example, if a function expecting a scalar input is called with a vector parameter, then the compiler will automatically vectorise the call. Thus, implementing an implied *for* loop, where the limits of the loop are automatically calculated. In situations where it is necessary to provide finer control of this automated unwrapping feature, all *classes* (and *aspects*) implement the *method* '*map*'. This in essence reduces the dimensionality of the input by one degree and calling in a vector fashion the designated *method* call. This is similar in concept to the "reduce" functionality central to functional programming [16]. Finally, as always, default type conversion behaviour can be overridden through the concepts of *aspects*.

An example of assigning different datatypes are as follows:

```
Integer16 var1;
Integer32 var2;
...
method() = doSomething() {
   var1 <- var2;
   ...
```

And an example of mapping vectors to smaller vectors is as follows:

```
Integer arrayOfNums[][] <- {{1, 2}, {3, 4}, {5, 6}, {7, 8}};

Integer anotherNums[] <- arrayOfNums.map(f, Integer(f));
```

where the mapping function 'f' is provided elsewhere.

3.7 Summary

In this chapter, we have shown some of the required features for signal and image processing applications on FPGAs. The coding expressiveness and efficiency have been shown with small examples. There are more issues to be considered such as the expression and implementation of signal processing algorithms using mixed data width. Interesting work in this direction has already been undertaken by the group at Imperial College [19,26] as well as research into resource allocation and optimisation for best use (in such terms as: processing power, space and power drain). This is particularly crucial for the new class of FPGA devices that include distributed complex resources such as multipliers with DSP function units and microprocessor cores fabricated onto them. A number of papers presenting the ability to tackle these sorts of imaging problems are presented in [4, 5, 9, 23, 27, 45].

The next chapter describes the compiler implementation process in general.

Chapter 4

Compiler

This chapter is an overview of a compiler design and writing process. The structure of a basic compiler can be split into three parts [1] — front-end, intermediate stage and back-end — as shown in Figure 4.1. One stage is completed and followed by the next stage to give complete modularity. The detail of each stage is described in the following sub-sections. This chapter concentrates on the front-end more than the intermediate and the back-end, while the detailed implementation of the front-end is covered in Chapter 5. The intermediate-stage implementation is covered in Chapter 6 and the back-end implementation is covered in Chapter 7.

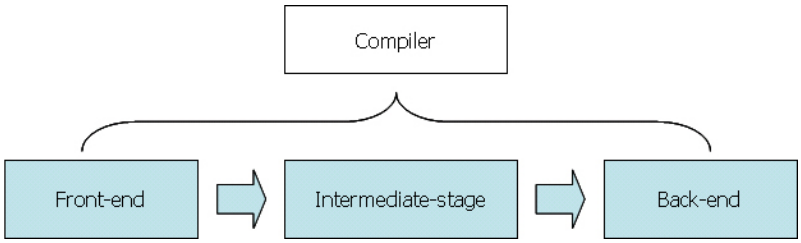


Figure 4.1: Compiling process

4.1 Front-end

The front-end of a compiler is shown in Figure 4.2. The main tasks involve recognizing tokens (lexical analysis) and syntactic analysis (parsing and syntax checking). This process is a well-understood problem hence there are many tools available,

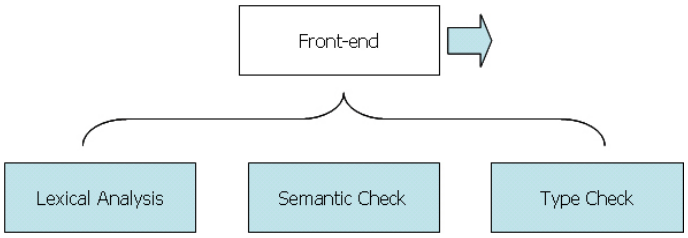


Figure 4.2: Front-end of a compiler

such as *lex*, *flex* and *CUP* [43, 51] for the lexical analysis and *yacc*, *bison* and *javacc* [30, 49, 51] for the syntactic analysis. As stated, most tools are broken into two programs, lexical analyser and syntactic analyser. The output of the lexical analyser is fed to the syntactic analyser and the output of the syntactic analyser is a parse tree that is used for the transformation to the control-flow graph in the intermediate-stage.

For implementation, we have used is *jjtree* [50], which is a pre-processor for *javacc*. The *jjtree* generates a simple parse tree (namely *SimpleNode*) after reading a program. One of the major reasons for using *javacc* over the others was that it produces *Java* outputs, which our compiler is implemented. The *javacc* also combines the lexical analyser and syntactic analyser. It is a powerful *LL(k)* parser [64] and provides a pre-processor that generates a parse-tree.

Lexical analysis is done automatically by a tool, *javacc*. It simply determines the tokens and feeds them to the semantic check, which then determines the usefulness of the token and whether it is a reserved keyword, a variable or a constant. The grammar of the language is written in BNF (Backus-Naur Form) form [1] and when the *javacc* detects mis-matches with the given grammar, it automatically produces error-messages and stops the compiling process if necessary. Some error recovery is handled by *javacc* and the compiling process can be continued when it finds one error; hence, it does not stop for each single error detection. In Figure 4.2, there are no arrows between one stage and the next stage because they are not done step by step. In our project, the semantic check and type check are implemented as an *aspect*, which is executed just-in-time by weaving. More detail of how *aspects* are used throughout the compiler implementation is discussed in Appendix D.

The full BNF of this language, ADH, is given in Appendix A. The main elements

involved in the front-end are parse-tree and symbol-table.

4.1.1 Parse-tree

The semantic check, done by *javacc*, generates a simple parse tree. An example of parsing `a + b` is shown in Figure 4.3. This tree is also called an "Abstract Syntax Tree" as the internal nodes are labelled by the operators and the leaf nodes contain the operands of the operators. This tree is extended so the nodes also contain some relevant information. In the example, 'AdditiveExpression' (which could be either + and −) becomes 'Add (+)' and the 'Identifier' has associated with name 'a', and we also discard the irrelevant information such as the empty multiplicative expressions. Therefore, we get an extended but compressed (or simplified) parse tree as shown in Figure 4.4. The unnecessary information is due to the design of BNF grammar, which is written in a way that detects mathematical precedence. For example, the compiler puts the additive expression into the stack first, followed by the multiplicative expression, followed by the unary expression. Hence, the unary expression could come out from the stack first and do its appropriate calculation followed by the multiplicative expression, and so on. These are the roles of semantic checks.

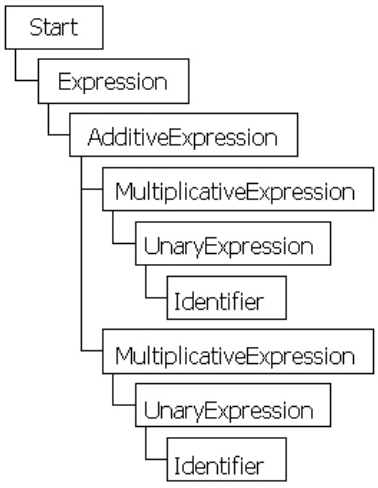


Figure 4.3: Creation of a simple parse tree for a grammar "a + b"

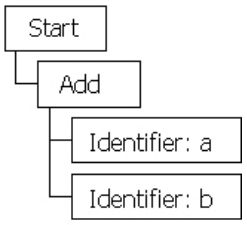


Figure 4.4: Creation of a simplified parse tree for a grammar "a + b"

4.1.2 Symbol-table

We also create a symbol table of declared identifiers. This is one of the major data structures used in a compiler that contains information such as the location in source code, type, scope and use of the symbols. The symbol table is managed in hierarchies and is in the form of a tree for our compiler. The root of this tree is called *concerns* and the next level is an object (*class* and *aspect*) followed by the *methods* and *class* variables, followed by the local variables. The attributes that we add to each node include visibility, scope of a variable, type of a variable, and definition of the variables. The symbol table is managed and modified in all of the three compilation processes by removing, adding and copying the nodes within the tree. For an example, the language in Figure 4.5 is read and then transformed into the symbol table in Figure 4.6.

```
1:  public class Simple {
2:      method (Integer result;} = Adder(Integer a; Integer b;) {
3:          result <- a + b;
4:      }
5:  }
6:
7:  class Integer { }
```

Figure 4.5: Example ADH code

4.1.3 Type checking

There are two major operations required in the front-end of a compiler: semantic check and type check. The semantic check basically checks that the meaning of a statement is correct. Typical examples of semantical information that is added

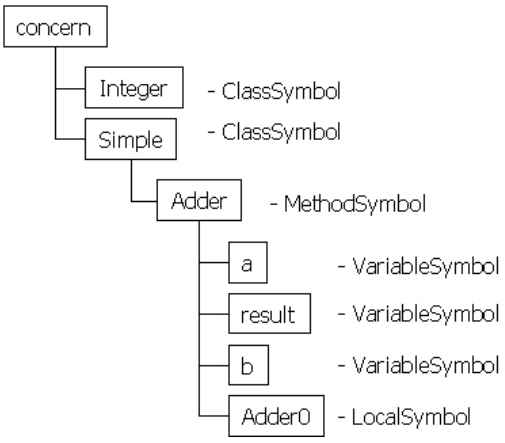


Figure 4.6: Example of a symbol table

and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding).

The type-check also falls into the semantic check but it basically checks for whether the assignments are done with the correct types. For example, syntactically, a variable could be assigned to another variable. However, it may not be operable if an *Integer* is assigned to a *String*, without any conversion or type casting. The type check automatically converts to the correct type when there is any mismatch. This might not always be possible and the compiler indicates exceptions where appropriate. Automatic type conversion is one of the useful features in that different lengths (in terms of bits) of constants can be assigned. In contrast, VHDL is a strong-typed language; hence, the developers are required to convert the types manually.

4.2 Intermediate stage

The intermediate stage starts its processing from the parse tree. The tree is converted into a control-flow graph (CFG[1]) [64]. The main purpose of the intermediate stage is to generate the machine independent intermediate language that makes it easier to produce the final machine dependent code. Therefore, the intermediate language (IL) can be translated to many different target machines, such as Xilinx

[1]Typically, CFG in front-end stands for 'Context-Free Grammar', but in intermediate-stage stands for 'Control-Flow Graph'

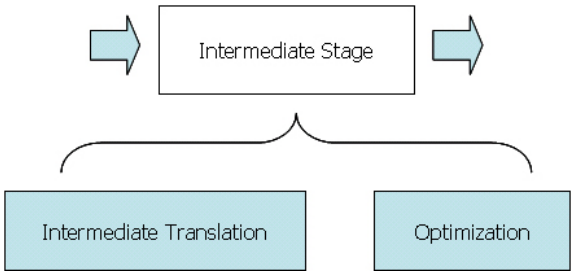


Figure 4.7: Intermediate stage of a compiler

Vertex FPGA or Spartan FPGA. Alternatively, the target machines could be a totally different architecture, such as microprocessors rather than FPGAs. The steps involved in the intermediate stage are outlined in Figure 4.7. The main elements used in the intermediate state are 'basic blocks' and 'control-flow graph', and they are defined in the subsections.

The intermediate stage also contains a number of optimisation techniques such as static single assignment form [20,64]. There are many optimisation techniques that can be performed in the intermediate stage. Local optimisation, global optimisation, dead code elimination and loop unrolling are examples of typical optimisation techniques, and the one we are mostly interested in is the SSA form.

4.2.1 Static Single Assignment

Many dataflow analyses need to find the use sites and the definition that the variable holds at that instance. The static single assignment (SSA) form encodes distinct definitions and uses for each variable. This SSA form does not have an obvious generation algorithm. Textbooks such as Appel [64] and Cooper [20] give some algorithms and we have followed the algorithm from Appel, which basically follows the process in Figure 4.8. The detailed theory and mathematics behinds this technique is not within the scope of this thesis; therefore, only the outline is provided here.

SSA form ensures that each static variable only has one definition or assignment. Each use of the variable refers to a single definition. This form is very useful in hardware because the dataflow in microprocessors can be controlled and changed by the stack frame and stack pointer; however, in FPGAs fixed hardware circuitry is implemented. The hardware device physically can have only a single driver. Multiple

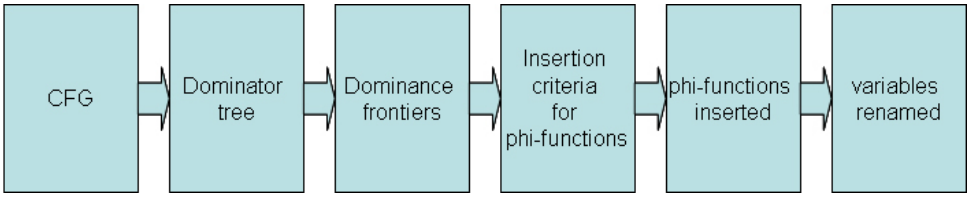


Figure 4.8: SSA form construction algorithm

drivers must be controlled by a multiplexer.

The transformation of IR into SSA form requires an insertion of the phi-function at points where control-flow paths merge and it renames variables to make the single assignment property hold.

Consider the small loop program,

```
x <- ...
y <- ...
while (x < 100)
    x <- x + 1
    y <- y + x
```

This is translated into the following SSA form.

```
      x0 <- ...
      y0 <- ...
      if (x0 >= 100) goto next
loop: x1 <- phi(x0, x2)
      y1 <- phi(y0, y2)
      x2 <- x1 + 1
      y2 <- y1 + x2
      if (x2 < 100) goto loop
next: x3 <- phi(x0, x2)
      y3 <- phi(y0, y2)
```

The phi-functions have been inserted at points where multiple distinct values can reach the start of a block. The 'while' command has been rewritten with two distinct tests ('if' and 'goto') and the initial test refers to 'x0' while the end-of loop tests refers to 'x2'.

Figure 4.9[2] shows another example of how SSA form gets generated from the source code.

[2]Figure adopted from *Modern compiler implementation in Java* by Appel [64]

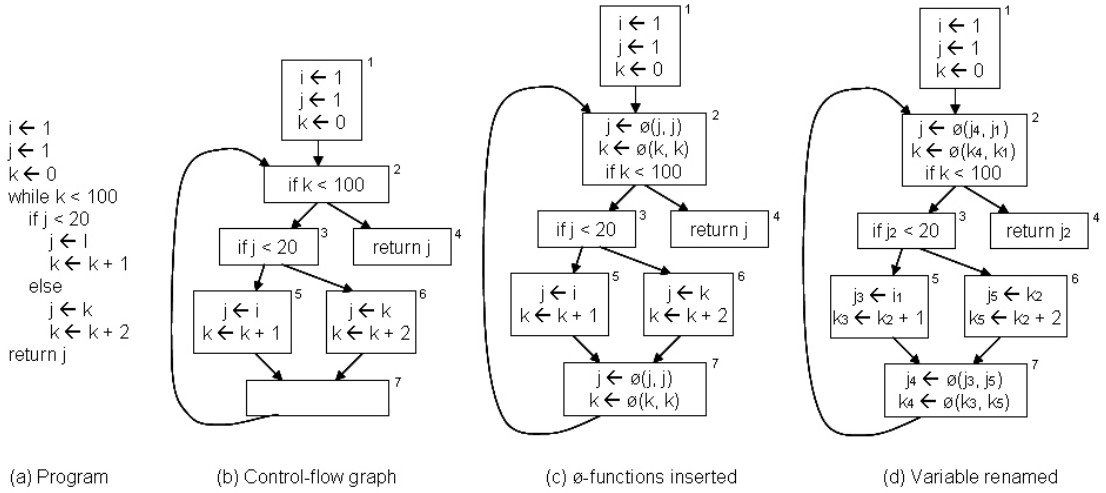


Figure 4.9: SSA form construction example

4.2.2 Basic-block

A basic block [64] is a straight-line piece of code without any jumps or jump targets in the middle. In determining where the jumps go in a program, an analysis of the control flow is required. A control flow is the sequencing of instructions in a program. Basic blocks are usually the fundamental unit to which compiler optimisations are applied. Basic blocks form the nodes in a control flow graph.

A basic block is a sequence of statements that is always entered at the beginning and exited at the end, therefore it has the following properties:

- The first statement is a LABEL
- The last statement is a JUMP
- There are no other LABELs or JUMPs

More formally, a sequence of instructions form a basic block and the instruction in each position dominates, or always executes before, all those in later positions, and no other instruction executes between two instructions in the sequence. The blocks to which control may transfer after reaching the end of a block are called that block's successors, while the blocks from which control may have come when entering a block are called that block's predecessors.

4.2.3 Control-flow graph

A control flow graph (CFG) [64] is an abstract data structure (or an abstract representation) of a procedure or program, maintained internally by a compiler. Each node in the graph represents a basic block and directed edges are used to represent jumps in the control flow. There are two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves, as shown in Figure 4.10. This is particularly useful in the hardware description language. The hardware devices are fully configured at the time of running; therefore, the statements can be executed or bypassed depending on the condition. This is further explained in Chapter 6.

A CFG contains number of basic blocks. It is a static representation of the program and represents all alternatives of control flow. So, for example, both arms of an *if* statement are represented in the CFG, as shown in Figure 4.11 (a). A cycle in a CFG may imply that there is a loop in the code (the cycle caused by a back edge to a dominator), as shown in Figure 4.11 (b).

The CFG is essential to several compiler optimisations based on global dataflow analysis (def-use chaining [64]). The 'def-use chain' models a relationship between the definition of each variables and their use. For each statement in the flow graph, the compiler can keep a list of pointers to all the *use* sites of variables defined there, and a list of pointers to all *definition* sites of the variables used there. In this way, the compiler can hop quickly from use to definition to use to definition.

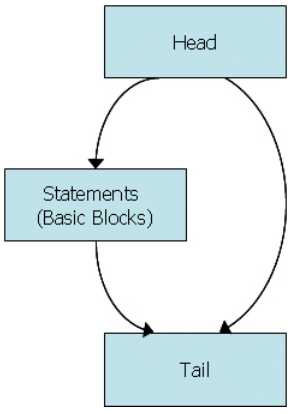


Figure 4.10: Intermediate-stage code-graph overview

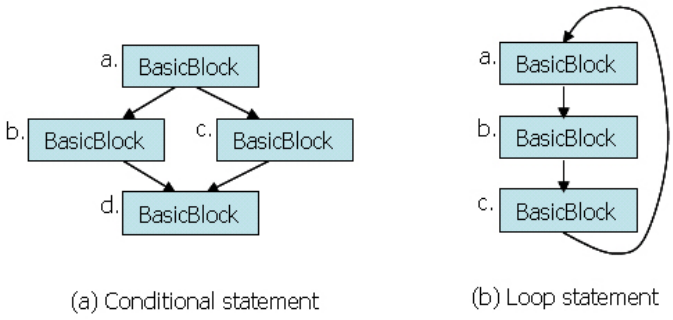


Figure 4.11: Different code-graphs generated by different statements

4.3 Back-end

The back-end generates the machine dependent final code, which is VHDL for this project. As outlined in the introduction, this project produces VHDL code to show the power and effectiveness of this new language, ADH. It is possible to qualitatively compare the generated VHDL code with ADH as well as compare it with hand-written VHDL codes, and find improvement points. It also provides the ability of debugging the compiler. The back-end can be broken into two major parts as shown in Figure 4.12.

The first part is code generation, which converts the basic blocks and IL to VHDL, and the other part is to make the external I/O pin connections. In the current FPGA development situation, the pin connection information is stored in a separate constraint file called 'ucf', and it can be done in either the CAD method as shown in Figure 4.13 or a text file editor to directly add information in the 'ucf' file. In our language, the pin connection can also be done as part of language; hence, this part of language needs to be converted to a 'ucf' file by the compiler.

The code generation in the back-end is the most difficult part in this compiler

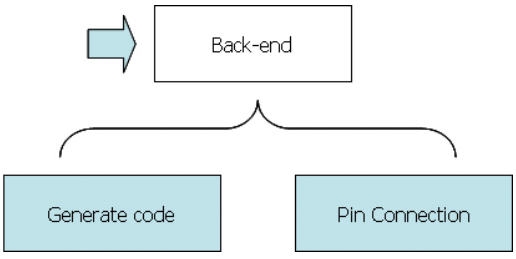


Figure 4.12: Back-end of a compiler

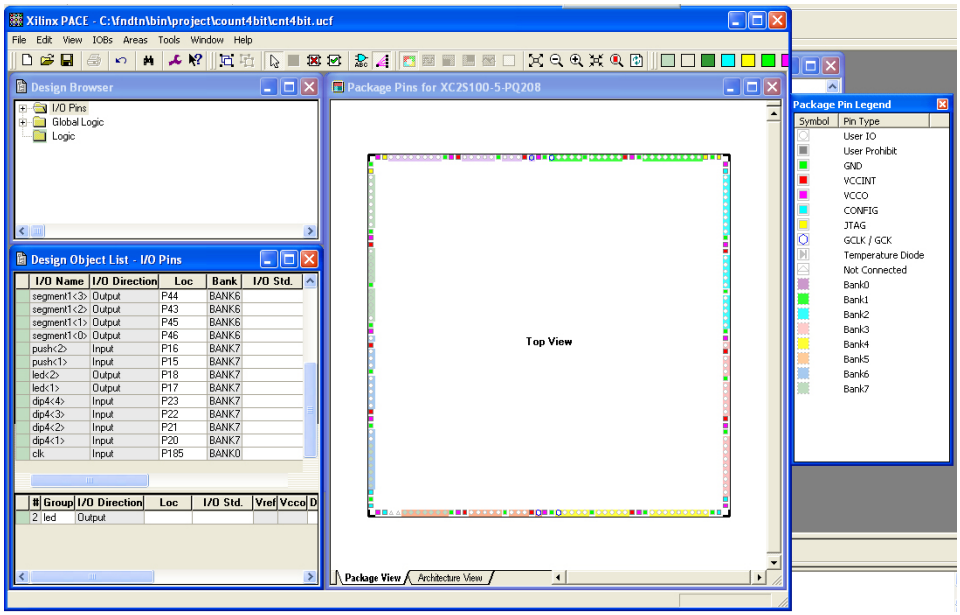


Figure 4.13: Xilinx Pin connection using PACE (Pinout and Area Constraints Editor)

writing process. The high-level concepts need to be converted into lower-level implementation, which causes some problems. Unlike the front-end, there are not any tools available for the back-end because results vary depending on the architecture of the target hardware.

The target code, VHDL, can be written in behavioural style as well as in structural style as described earlier. However, the top-most level is usually written in a schematic diagram with the help of CAD tools because it is easier. Alternatively, it can be written in a structured style of VHDL, which is similar to the schematic diagram. Hence, the *methods* in ADH are written in a behavioural style of VHDL and the *classes* in ADH are written in a structured style of VHDL. The detailed implementation method is discussed in the following chapter.

Chapter 5

Front-end and Weaving Implementation

In this chapter, we explain the detail of the front-end implementation of the compiler and the weaving technique, which is performed in a part of the front-end. The use of design patterns is important to this implementation; in particular the visitor pattern is the most notable one used to create and to manage the symbols. The design patterns can be found in many books [29, 32, 63] and the ones we have used are briefly described in Appendix D.1.

Before going into the detail of the compiler implementation, it should be clear to the reader that the language ADH supports aspect-oriented features and the aspect codes are spread to the multiple object through the *weaving* process. However, we have also used an aspect-oriented programming language, *AspectJ*, for the compiler writing and *AspectJ* uses its own *weaving* process to spread my codes into multiple objects. The two *weaving* processes are not the same and the reader should clearly understand the difference between the two that we are referring to.

5.1 Front-end Implementation

The design of a symbol table and its element takes considerable effort as the management of the symbols are very important throughout the three compiler stages. The front-end and weaving implementations are described with examples.

5.1.1 Symbol table generation

The symbol table is built at the same time that the parse tree is built by the parsing process. The defined variables are added into the symbol table with the symbol property shown in the symbol hierarchy. An example of a symbol table was shown in Figure 4.6.

The symbol table is constructed in a tree-like structure. The 'MethodSymbol' in the symbol table has a link to a parse tree in which the ADH statements are constructed in the form of a tree. Figure 5.1 shows an example of a symbol table and a parse tree generated by the parsing process. The source code that generated this example is the same as the one in Figure 4.5.

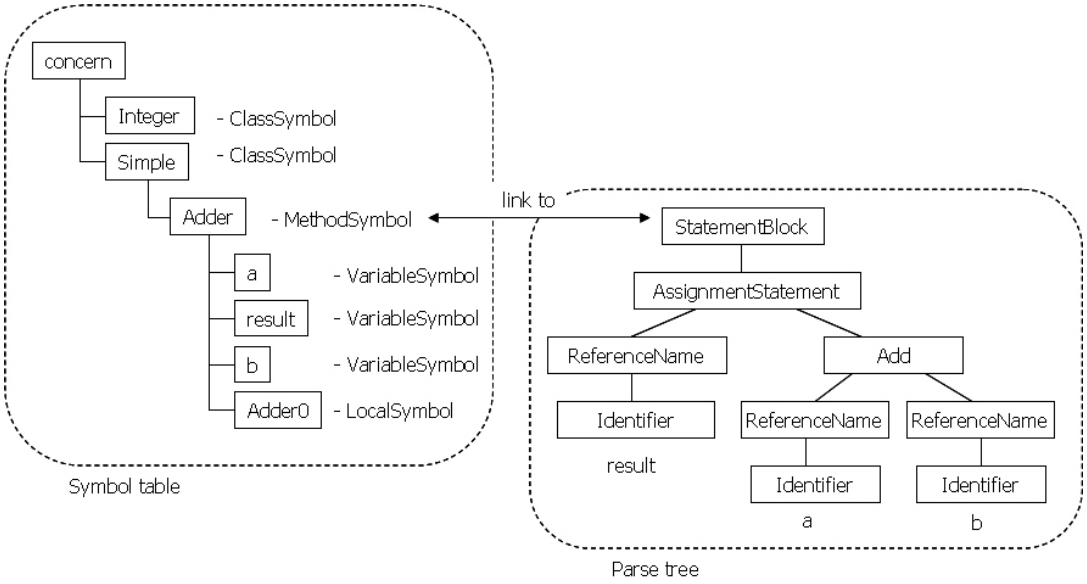


Figure 5.1: An example of a symbol table and a parse tree

5.1.2 Symbol management

A symbol is a representation that refers to a variable or a function or a data type in the source code. The management of these symbols is very important because the symbol table is heavily accessed during the entire compilation process. Some of the operations are required to be applied to a group of symbols.

Figure 5.2 is the UML (Unified Modelling Language) [28] diagram of symbols and it shows how symbols are managed. This hierarchy is different to the hierarchy

of the symbol table itself. It only shows the management of the symbols. For example, we might want to apply some actions to 'EntitySymbol', and the type and the value information can be retrieved only from the entity symbols (constant symbol, identifier symbol, variable symbol) but not from the others.

The 'FunctionSymbol' have input and output parameters that are irrelevant to the other symbols. These common functionalities are written in the higher hierarchy with the 'strategy pattern' [29] and the lower hierarchy symbols could use the common attributes and functionalities through the use of *inheritance*.

The database symbols, 'ConcernDB' and 'InstanceDB' are created using the 'singleton pattern' [29]. The instantiation of the 'ConcernDB' symbol is named 'concerns'. It is the root of the symbol table and is where each of the three compilation process starts.

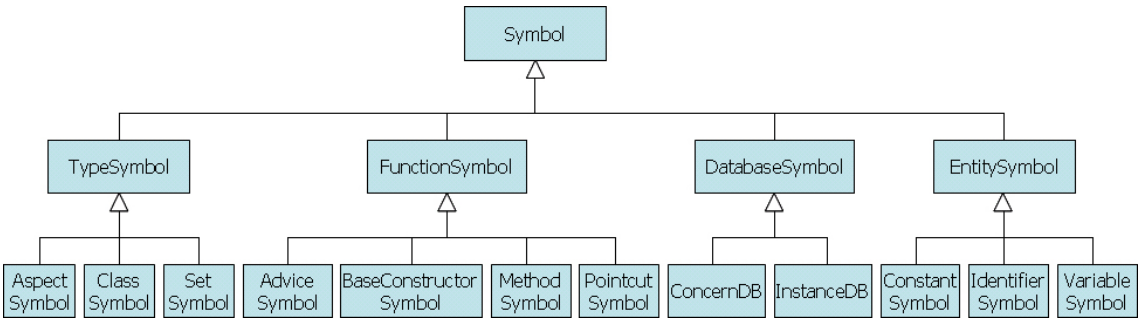


Figure 5.2: Symbol management hierarchy

5.1.3 Operations in the front-end

Once the symbol table is built, we need to apply some actions to the symbol table, for example, a semantic check. This requires traversing the symbol table and accessing the symbols. The major operation in the front-end is the semantic check. However, for clarity, type check and reference check are implemented in separate modules. There will be more operations in the intermediate stage that process the symbols, which are not all known at the time of writing the front-end stage. It has motivated us to use the 'visitor pattern' [32] that provides a very elegant solution in this case.

The problems with the ordinary object-oriented visitor pattern and the implementation of the new aspect visitor pattern are more fully explained in Ap-

pendix D.1.2.

Figure 5.3 shows the first implementation of the symbol hierarchy and the operations required in the front-end using the visitor pattern. The symbol hierarchy shown is the same as Figure 5.2 but now there are more classes on the right-hand side that traverse the symbol table and add some actions to the symbols.

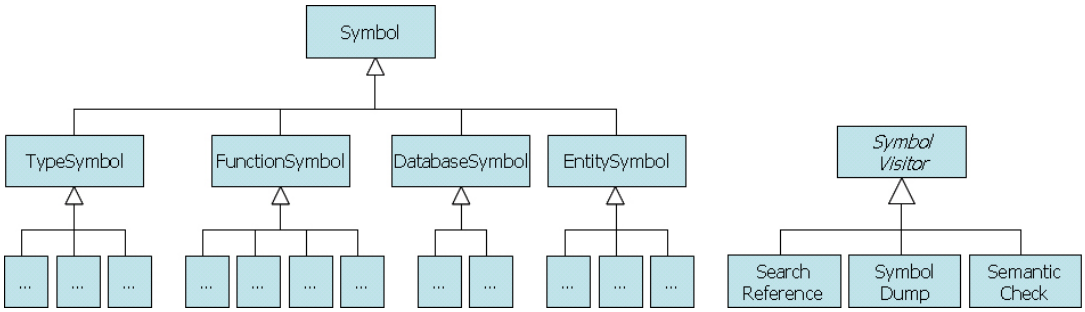


Figure 5.3: Implementation of the visitor method on symbols hierarchy

The visitor pattern contains the following behaviours:

- SearchReference: Find the reference to a symbol
- SymbolDump: Display the hierarchy outputs on screen for debug purposes
- SemanticCheck: Check for the syntactic correctness

Additional functionality can be added under the 'symbol visitor' because the symbol hierarchy is designed to support the visitor pattern. For example, the intermediate code translation can also be added to this visitor pattern.

However, the problems with the ordinary visitor pattern — the awkward double calling mechanism and also the possibility of runtime exceptions — has led to the implementation of the aspect visitor pattern. It provides code that is easier to understand, easier to read, and easier to maintain.

In this case, the symbol hierarchy does not have to have the *accept()* supporting *method* required for the ordinary visitor pattern. The code become cleaner and this reduces the *coupling* between the classes and effectively increases the *cohesion*. The aspect visitor pattern uses *introduction* to implement the visitor pattern and the net results are the same as the ordinary visitor pattern.

Figure 5.4 shows the partial implementation of the 'aspect visitor' implementation. It shows 'Semantic check', 'Type check' and 'Aspect weaving' written using *AspectJ* and the *weaving* will spread the code into the multiple objects (symbols).

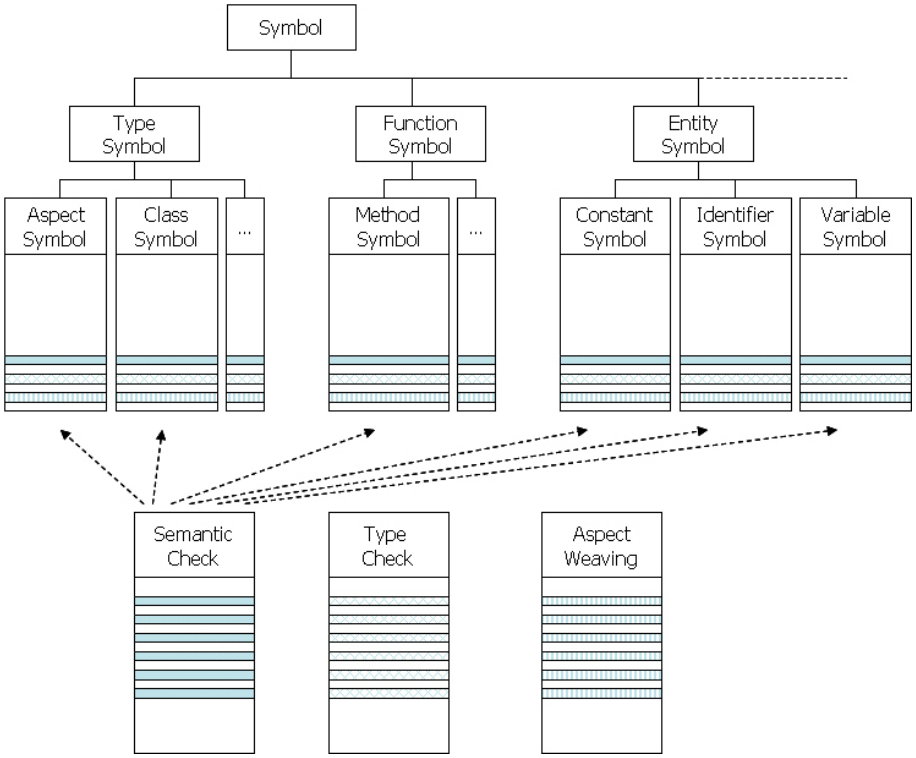


Figure 5.4: The usage of AspectJ for the operations in the front-end

5.2 Weaving Implementation

The compiler, using a process known as *weaving*, modifies the execution control graph to instantiate an *aspect* into the program. The different *weaving* techniques are briefly outlined in Chapter 3.2 and we have used the compile-time *weaving*. It acts similarly to the pre-processor with the parse-tree manipulation occurring at the front-end, just before the semantic check starts. In this case, no separate semantic check is required for aspect support. This process is outlined in Figure 5.5.

This pre-processor approach is only possible for the static features of the aspect, but that is what we are concerned with at the moment. Implementing dynamic features in hardware are also not feasible. Unlike with microprocessors where the execution point can be jumped and controlled, the FPGA circuitry is fully configured

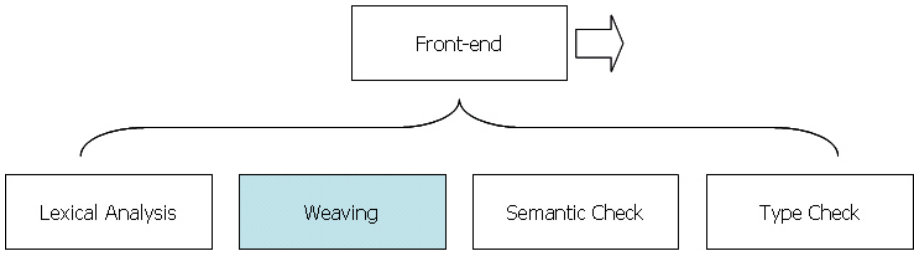


Figure 5.5: Weaving process occurring in the front-end

at the time of running. Entire code is being executed concurrently and there is no jump or dynamic control determination. Therefore, the dynamic features must be determined and defined for hardware. Even though we can imagine dynamic hardware and there is a hardware device that allows dynamic configuration, they are not within the scope of this thesis.

The *aspect* features that we are concerned with are the static cross-cutting elements such as the *introduction*. As the name 'static' suggests, these can be identified and the location (join point) of the *weaving* can also be found at compile-time. There is not any work involved in run-time; therefore, the aspect could be resolved into the designated object before processing to the intermediate stage.

Figure 5.6 shows an example of a simplified parse-tree, which has one *class* and one *aspect*, generated after completion of the front-end parsing processing. Before proceeding to the intermediate-stage, we do a static weaving that can be described in four steps.

The compiler:

1. Scans the parse tree to find the *aspects*
2. Looks at the *advice* and finds the *pointcut* that contains information on where it needs to be applied to the *class*
3. Select the executable statements that are below *AdviceSymbol* and insert them where appropriate by looking at the *advice*
4. Updates the symbol table if necessary

The *advice* contains one of three location informations — *before*, *after* or *around*. If it is *before*, it inserts the contents at the top of the designated place (i.e. head of

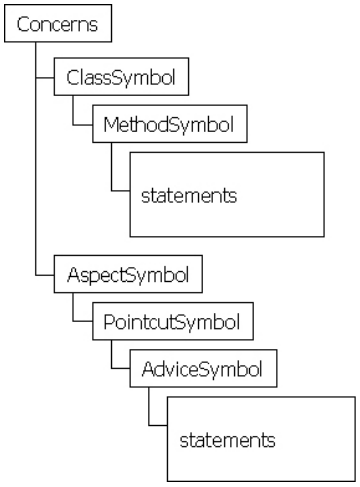


Figure 5.6: A simplified parse tree generated after the front-end

the tree), if it is *after*, it inserts the contents at the bottom of the place (i.e. tail of the tree), and if it is *around*, then it simply replaces the existing contents with the aspect contents. Figure 5.7 shows how the compiler finds the place where the statements are to be added through the weaving.

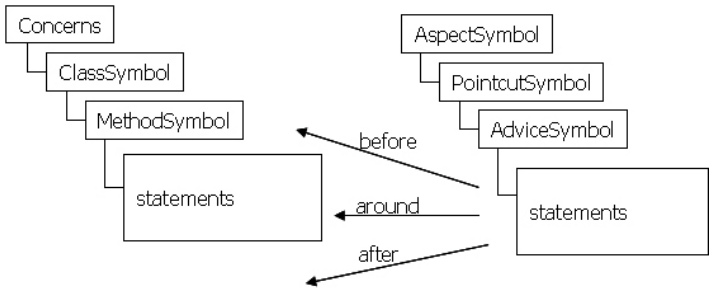


Figure 5.7: Weaving process finding its location

Figure 5.9 shows the actual implementation and the result of the static introduction of a variable added to the tail with *after* advice, with the source code shown in Figure 5.8. This introduces a new variable to an existing class; therefore, modification of the parse tree requires the update to the symbol table as well.

The example basically shows that for each addition of two numbers, the *aspect* also calculates the difference of the two numbers. This may be trivial and it could be argued that the subtraction *method* can just be written as a *class* object rather than an *aspect*. However, this is just an illustration showing that the aspect can be used to extend the existing *class*, and more powerful *weaving* such as replacing the

```
 1:  class Simple {
 2:    method(Integer sum;) = Adder(Integer a; Integer b;) {
 3:      sum <- a + b;
 4:    }
 5:  }
 6:
 7:  aspect Test {
 8:    Integer Simple.diff <- 0;
 9:
10:    pointcut catchAdd()
11:      : call(method (Integer result;) = Simple.Adder(Integer a; Integer b;));
12:
13:    after() : catchAdd() {
14:       Simple.diff <- a - b;
15:    }
16:  }
```

Figure 5.8: Source code for static introduction of variables

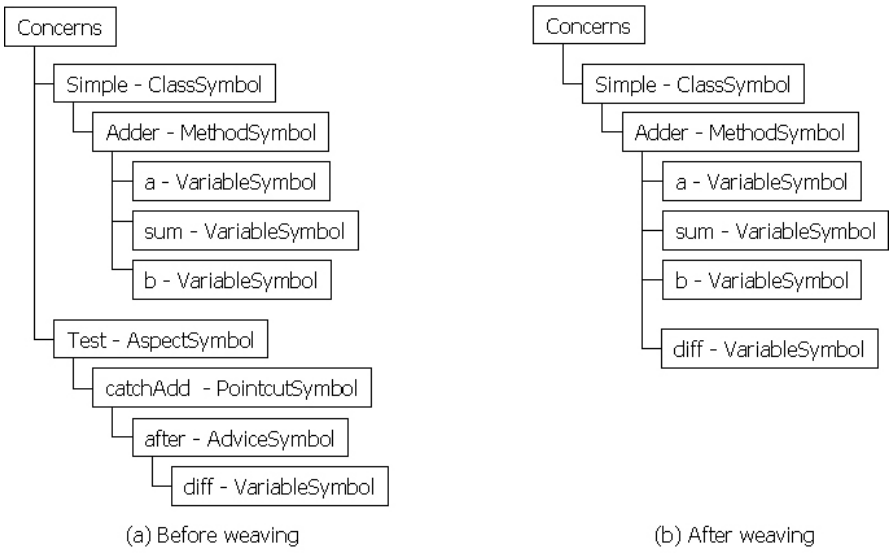


Figure 5.9: Weaving process for a static introduction of a variable

```
 1:  class Simple {
 2:    method(Integer sum;) = Adder(Integer a; Integer b;) {
 3:      sum <- a + b;
 4:    }
 5:  }
 6:
 7:  aspect Test {
 8:    pointcut catchAdd()
 9:      : call(method (Integer sum;) = Simple.Adder(Integer a; Integer b;));
10:
11:    around() : catchAdd() {
12:       sum <- OptClass.optimizedAdd(a, b);
13:    }
14:  }
```

Figure 5.10: Example of swapping the code for optimisation

existing adder code with hardware optimised code (such as 'carry look-ahead adder' instead of 'sequential adder') is also shown in Figure 5.10.

This simple technique can be called 'code weaving' and has the limitation that it cannot be extended to the advanced dynamic cross-cutting that determines the control sequence at run-time; however, this has the following benefits:

- It is easy to code and debug — the intermediate woven code is visible to the debugger.
- The scope is the static cross-cutting and this is faster and more efficient with 'code weaving'.

5.3 Summary

In this chapter, we have shown the front-end implementation that manages the symbols and produces the symbol table. A useful algorithm such as an aspect visitor pattern has been used and is fully described in Appendix D.1.2 for clarity. This chapter only focus on the important front-end and weaving technique that we have used, not the particular software implementation techniques. The *weaving* is implemented as a part of the front-end and it effectively resolves the static feature

of aspect-oriented support. The following chapter presents the implementation of the intermediate-stage.

Chapter 6

Intermediate-stage Implementation

The completion of the front-end process gives a parse tree and a symbol table. The aspect features are all resolved by the front-end *weaving*. At this stage, the parse tree contains syntactically and semantically correct information. This parse tree is still closer to the source code and basically it is just a list of statements represented as a tree. During the intermediate-stage, this tree is broken into several basic blocks and the one-directional parse-tree becomes a bi-directional graph, which is termed a control-flow graph. The generated graph contains intermediate language (IL), which is a machine independent code, and intermediate representation (IR) and they help dataflow analysis for optimisations.

The intermediate-stage can again be broken into two smaller stages. The first stage is the process of parse-tree transformation into the control-flow graph and the second stage is the optimisation. The control-flow graph is the abstract idea and our implementation of this control-flow graph is called a code-graph. The code-graph contains a number of basic blocks containing primitive statements (IL) put together in an IR form. The code-graph is the main intermediate language. In this chapter, we will present the generation of this code-graph in detail with examples.

6.1 Intermediate-code generation

The *methods* in OO are the core part of the language containing how the procedures are executed and what the outputs are. Written in ADH, the statements outside the *methods*, such as the statements defining the *class* and *methods* itself, together make up the shell of the *class* and *methods*, and these define the scope and the visibility, which is shown in Figure 6.1. These shell statements are already processed and stored in the symbol table. Hence, we now only need to process the *methods* statements into the intermediate-code, which contains a list of statements that need to be executed.

There are two major processes in the intermediate-stage. One is the translation of the parse-tree into the IR that organises the basic blocks to produce a control-flow graph (CFG), and the other is the translation of the ADH statements into seven primitive statements, IL. These two translation processes occur in parallel, but we will explain the generation of the seven primitive statements first.

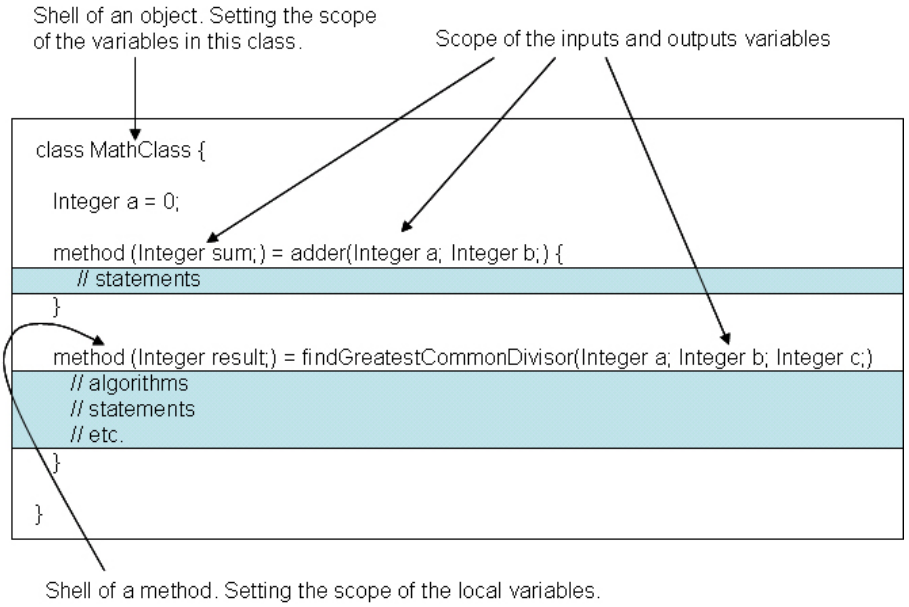


Figure 6.1: An object representation with the core statements highlighted in the gray bands

6.1.1 Statements translation

The IL that we use converts every statement into either instantiation of a function or a function call that has two operands, similar to the three-address-code [1] that is one of the common ILs that has been used for many years. The reason we treat everything as a function is that this makes the back-end process easier because the functions can become individual entities in VHDL. The call of a function will simplify instantiation of the hardware components in VHDL and this function call will proceed with the use of the instantiated piece of hardware.

A longer statement that has more than two operands will simply create temporary variables like the three-address-code. The following example illustrates how the statement is broken into groups of two operands:

```
a <- b * (c + d);
```

into,

```
t1 <- c + d;
t2 <- b * t1;
a <- t2;
```

The statements in ADH language are converted into one of the following seven statements — *InstanceStatement, CallStatement, EquateStatement, MultiplexerStatement, ConditionalStatement, InConnectionStatement* and *OutConnectionStatement*. Figure 6.2 is a UML diagram showing these statements but it shows eight primitive statements. The one additional statement, *ResolutionStatement*, added by the SSA optimisation technique is explained in a later section.

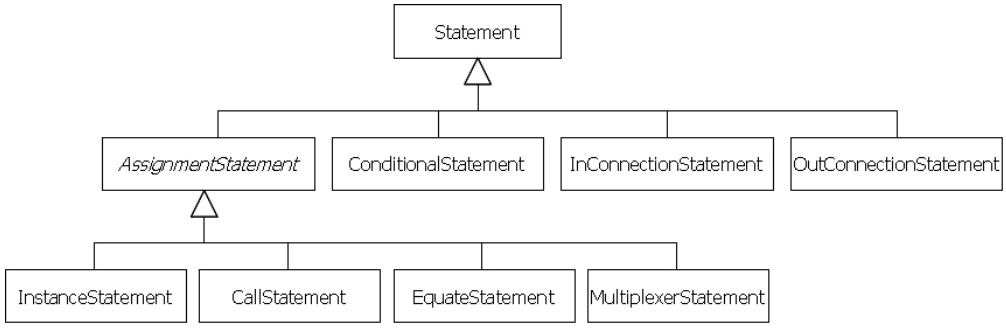


Figure 6.2: UML diagram for primitive statements

There are many keywords and control structures that can be described using ADH. All of these are turned into simpler statements that call library functions. These statements are the smallest unit; therefore we call them primitive statements. Limiting to a lower number (eight primitives in total) makes the back-end translation easier. These eight primitives act like instruction sets in microprocessors. Every function in a language is a call to another designed function or supported by the built-in library functions. Simple arithmetic instructions 'add', 'multiply' etc. and comparator operators such as '<', '=' are also supported as library functions with names 'lessThan()' and 'equalTo()' respectively.

The following explains, with examples, how the seven primitive statements are generated.

InstanceStatement

When a new function is created (instantiated) or a library function is called, then *InstanceStatement* is used. For example, addition of the two variables,

```
a + b;
```

yields creation of new 'Add' function, therefore generating the following *InstanceStatement*,

```
t@1 <- new Add()
```

This *InstanceStatement* is always followed by the *CallStatement* and the generated temporary variable '`t@1`' is used in the *CallStatement*.

CallStatement

Whenever a function call is made such as a *method* call or the operations, then *CallStatement* is used. The same example applies to the *CallStatement*,

```
a + b;
```

yields a function to be called by the following *CallStatement*, which exists in the instance piece of hardware produced by the prior *InstanceStatement*,

```
t@1.add(a, b)
```

In the end, the addition of the two variables (‘a’ and ‘b’) requires the instance of the ‘Add’ module by the *InstanceStatement* and the use of the instantiated hardware by the *CallStatement*.

A complex statement (for example, state-machine control structure) can also be broken into primitive statements by the use of *InstanceStatement* and *CallStatement*. For example,

```
state (stateVar, clk) {
   Fred -> Jack when (result == 2);
}
```

yields to the several ‘equalTo()’ and ’And()’ library functions,

```
t@a <- new EqualTo(),
t@b <- t@a.equalTo(statevar, Fred),
t@c <- new EqualTo(),
t@d <- t@c.equalTo(result, 2),
t@e <- new And(),
t@f <- t@e.and(t@b, t@d),
t@10 <- new And(),
t@11 <- t@10.and(t@f, clk),
```

All of the high-level language description or the complex control structure can be simplified to the primitive instructions that call supporting library functions.

EquateStatement

For the assignment of variables, *EquateStatement* is used. For example,

```
result <- a + c;
```

yields the following *EquateStatement*,

```
result <- t@2
```

where ‘`t@2`’ has been calculated for the addition of the two variables ‘a’ and ‘c’ by the use of *InstanceStatement* and *CallStatement* as per the previous examples.

MultiplexerStatement

When there are multiple assignments to one variable, this requires a hardware multiplexer. Then, this *MultiplexerStatement* is used. For example,

```
result <- a+b when (statevariable == Fred)
          a+c otherwise;
```

yields the variable 'result' is driven by the two different statements, the sum of 'a' and 'b', and the sum of 'a' and 'c'. It yields the following *MultiplexerStatement*,

```
t@7 <- Multiplexer (t@2, t@6 :: t@4)
```

where '`t@7`' is assigned to the 'result' at the end, and '`t@4`' is the control statement determining which input should be selected. The '`t@2`' is the result of '`a+b`' and the '`t@6`' is the result of '`a+c`'. There can be more than two choices which can lead to the following *MultiplexerStatement*.

```
t@10 <- Multiplexer (t@1, t@2, t@3, t@4 :: t@5)
```

where '`t@5`' is the control selector.

ConditionalStatement

The conditional statements "*if, then, else*" are *when* and *otherwise* in ADH. This uses the *ConditionalStatement*. For example,

```
when (a == b) {
   result <- a*b;
}
```

yields the following *ConditionalStatement*,

```
t@4 <- t@3.equalTo(t@1, t@2),
Test(t@4)
```

where '`t@1`' is the assignment of 'a' and '`t@2`' is the assignment of 'b'. The control structure can again be broken into the *InstanceStatement* and the *CallStatement*. The '`Test(t@4)`' is just a dummy variable making easier tracking of where the branch has been created and where the merge point is. The *ConditionalStatement* creates branches in the basic blocks; the control graph terminates the current basic block causing to fork into two branches (*true* and *false*) before merging back again.

InConnectionStatement

Used for an input pin assignment. For example,

```
var1 <- connection ("Pin13")
```

yields the following *InConnectionStatement*,

```
var1 <- Connection
```

OutConnectionStatement

Similar to the *InConnectionStatement*, an output pin assignment statement uses an *OutConectionStatement*.

```
connection ("Pin21", "Pin22", "Pin31") <- result;
```

yields the following *OutConnectionStatement*,

```
Connection <- result
```

6.1.2 ResolutionStatement

The optimisation techniques are performed in the intermediate-stage. The optimisation techniques are not within the scope of this thesis; however, the generation of static single assignment (SSA) form is essential. The SSA form helps a lot with dataflow analysis and optimisation. However, we use the SSA form not only for the optimisation but also to make the hardware instantiation realistic by adding a phi-function, which helps the multiple driver problems. The SSA form allows generation of the target VHDL code effectively. The hardware device can only have a single driver, hence the SSA form ensures that each variable has only one definition and is assigned only once. This has been described in Chapter 4.2.1.

At the end of SSA form generation, the code-graph contains unique variable definitions and adds a phi-function at the merge point of the conditional branches. Therefore, the phi-function statement is needed and we add one more statement making a total of eight primitive statements, including the *ResolutionStatement*.

The SSA form creates a resolution function to merge the branch. This effectively creates a multiplexer in VHDL. At the merge point, such as the branch merge generated by '*when* and *otherwise*', the following *ResolutionStatement* is generated,

```
t@4 <- ResolutionFunction (t@1, t@2, when t@3)
```

Therefore, depending on the condition of '`t@3`', '`t@1`' is assigned to '`t@4`' if the condition is satisfied, otherwise '`t@2`' is assigned to '`t@4`'. This is similar to the *MultiplxerStatement* and the SSA form effectively eliminates the multiplexer statements by assigning the variables only one unique value.

The two branches *true* and *false* generated by the *if*-like control structure are executed in the hardware regardless of the condition. The *ResolutionFunction* at the merge point then chooses which value to take on; therefore, this is particularly effective in the merging of the branches.

6.1.3 Basic-block generation

These primitive statements are managed in a group, called a basic-block. A basic-block is a data structure that contains a list of IL statements, but it also contains a lot of linking information. The linking information in the basic blocks is stored for the dataflow analysis and it helps generation of the SSA form.

Single source code statements can be turned into one or more primitive statements, and depending on their complexity, they can be grouped in one or more basic-blocks. The statements in a single basic block are processed in parallel. If there is any data dependency, they cannot be processed in parallel; hence, they must be broken into two or more basic blocks and processed one after the other.

Basic assignment statements are grouped into one basic block since there is no dataflow analysis required, but a complex statement such as conditional statements must be broken into four basic blocks as shown in Figure 6.3. In the figure, the top 'a' basic block contains comparison statements, the 'b' basic block contains

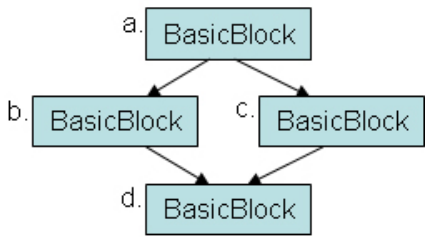


Figure 6.3: Basic blocks for conditional statements

```
source:
   a <- c + 2;
   res <- b + c;

primitive statements:
   t@1 <- new Add()
   t@2 <- t@1.add(c, 2)
   a <- t@2
   t@3 <- new Add()
   t@4 <- t@3.add(b, c)
   res <- t@4
```

Figure 6.4: Source codes transformed into primitive statements that make a basic block

the statements followed by *when* (*true*), the 'c' basic block contains the statements followed by *otherwise* (*false*), and the 'd' basic block is an empty merge point.

Figure 6.4 shows the two source code statements turned into a single basic block consisting of six lines of primitive statements. There are three types — *InstanceStatement*, *CallStatement* and *EquateStatement* — used from the eight primitive statements described in the earlier subsection. Since there are no data dependencies, they could be grouped into one basic-block and executed in parallel.

The entire *method* is turned into a group of these basic blocks as shown in Figure 6.5. The figure contains two conditional statements. Each basic block is connected together bi-directionally, using a double linked-list. It has a successor-list, which contains forward (downward) nodes, and a predecessor-list, which contains backward (upward) nodes. These linking information is useful for the optimisation techniques.

The collection of basic blocks form a code-graph that is described in the next section. The code-graph contains a head, statements, and a tail. These are reflected in Figure 6.5 with the statements being the entire middle basic blocks.

6.2 Code-graph generation examples

The generation of the IL statements and the organization of the basic blocks have been explained in the earlier sections. Here, we present the generation of a code-

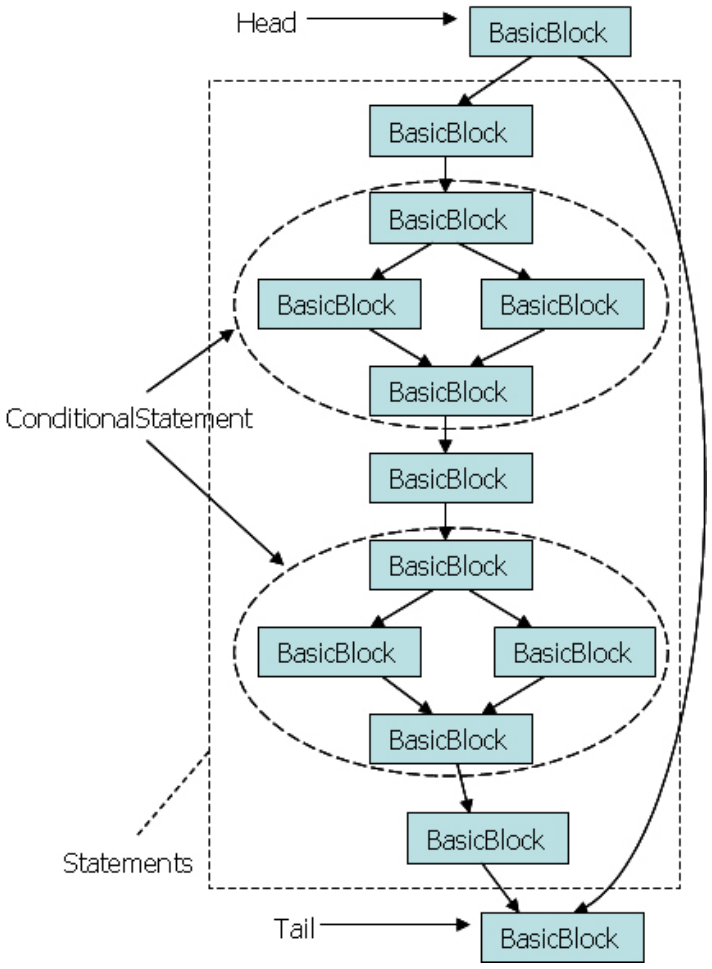


Figure 6.5: Example of a typical code-graph consisting of conditional statements

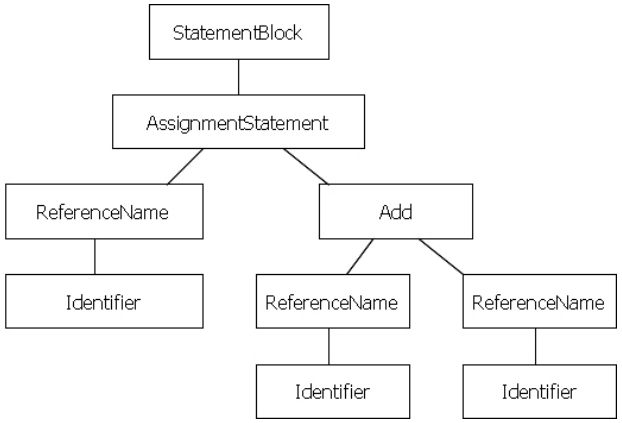


Figure 6.6: Statement block for "result < − a + b"

graph consisting of the IL statements and the basic blocks, with examples.

The intermediate-stage looks for the 'MethodSymbol' in the parse tree, which has a node called a 'statement block'. This has a link to a sub-tree that contains statements; for an example see Figure 6.6. The main task is to convert these statement blocks into a code-graph.

The statement shown in Figure 6.6,

```
result <- a + b;
```

is converted as follows:

```
t@1 <- new Add()
t@2 <- t@1.add(a,b)
result <- t@2
```

The converted code-graph can also be translated as follows, and these form a basic block:

```
instanceStatement
CallStatement
EquateStatement
```

A plain English version is that the execution of "`result <- a + b`" requires instantiation of an 'Add' module that is either generated in another package from another class or supported by a library function. For the particular example, it is the latter. The instantiation of the 'Add' package has an 'add' function (or entity) that takes two input signals and the result is equated to a variable called 'result'.

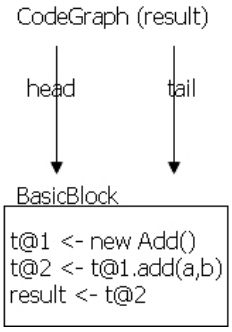


Figure 6.7: Code graph for a statement "result $<-$ a + b"

We have effectively created a code-graph that has one basic block, as shown in Figure 6.7. In the figure, the single statement is converted into a single node basic-block resulting in both the head and tail pointers pointing to the single code-graph. Multiple statements that have data dependency or a conditional control structure result in the tail pointer pointing to different basic-blocks, as shown in Figure 6.8.

The next step is to add a header (head) and footer (tail) forming a control-flow graph as shown in Figure 6.9, as described earlier in Chapter 4.2.3. A control-flow graph starts processing from the head and finishes at the tail. The statements can be executed or they can be by-passed depending on whether the module is called or not. In microprocessors, the *jumps* can be used to bypass certain operations but not

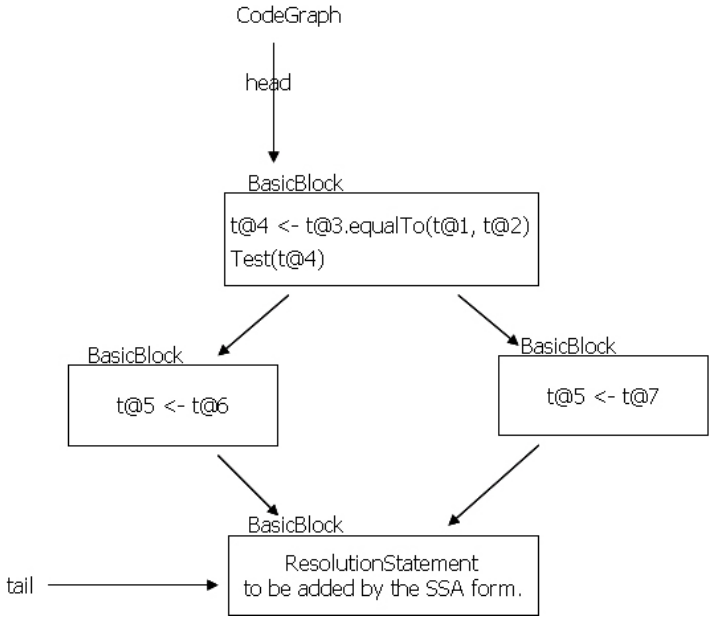


Figure 6.8: Code graph for a conditional statement "when ... otherwise ..."

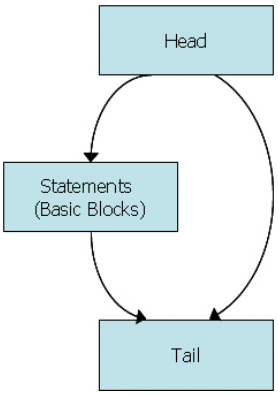


Figure 6.9: A code-graph in concept

in FPGAs. The circuitry in the hardware must be complete at the time of running. Therefore, both arms of the branches are executed and the resolution function added at the merge point chooses which value to take from one of the branches.

By adding the top and bottom nodes to Figure 6.7, we now have effectively created a full control-flow graph as shown in Figure 6.10. The resolution function is to be added to the bottom basic block by the SSA form generation. There is some more linking information in the basic blocks, such as the dominator frontier. They are not described in this thesis and, for clarity, are not shown in the figures but they are used for the SSA form generation.

6.3 Summary

We have shown how the intermediate-code is generated in this chapter. There are two separate processes in the intermediate-stage: the generation of a control-flow graph and the translation of the parse-tree into the intermediate language. They have been fully described in this chapter with examples. In the next chapter, the back-end, presents the translation of these intermediate languages into the final target machine codes. There are also important software implementation techniques that we have used to generate the code-graph and the basic blocks, such as the use of *AspectJ*. These implementation techniques are described in Appendix D.2 for clarity.

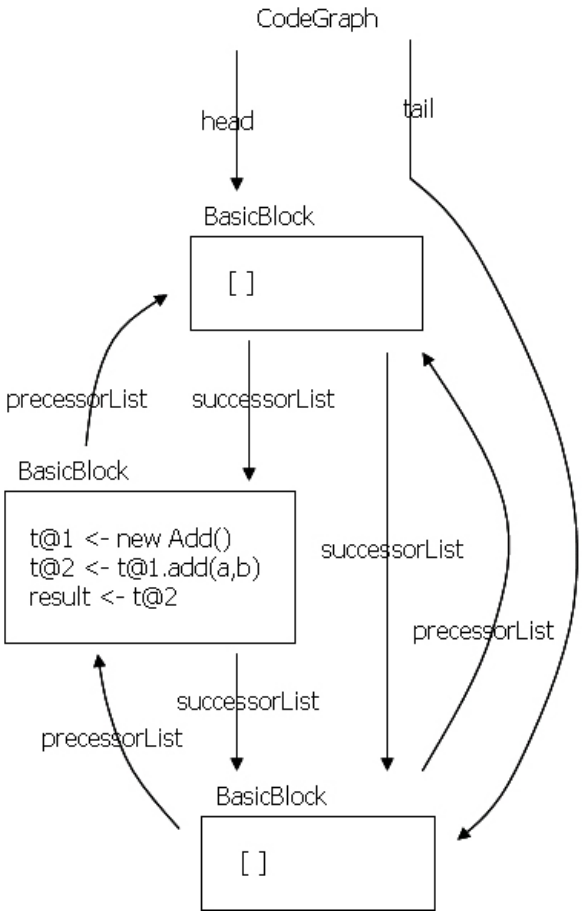


Figure 6.10: Code graph for a *method* that adds two variables

Chapter 7

Back-end Implementation

The back end of this project is VHDL code. From the intermediate stage, a code-graph has been generated. Every statement in ADH is turned into one of eight primitive statements implemented as function calls. Figure 7.1 is an overview of the compiling stages. Stage 3 in the figure is the intermediate translation and Stage 4 is the back-end process. We now need to convert these function calls and eight primitive statements into equivalent VHDL codes. The style of the VHDL code is structural where possible. Using the behavioural style would highly couple us to the VHDL language itself, making direct production of netlist form difficult in the future.

The front-end concentrated on the symbol table and it mainly focuses on the *class*

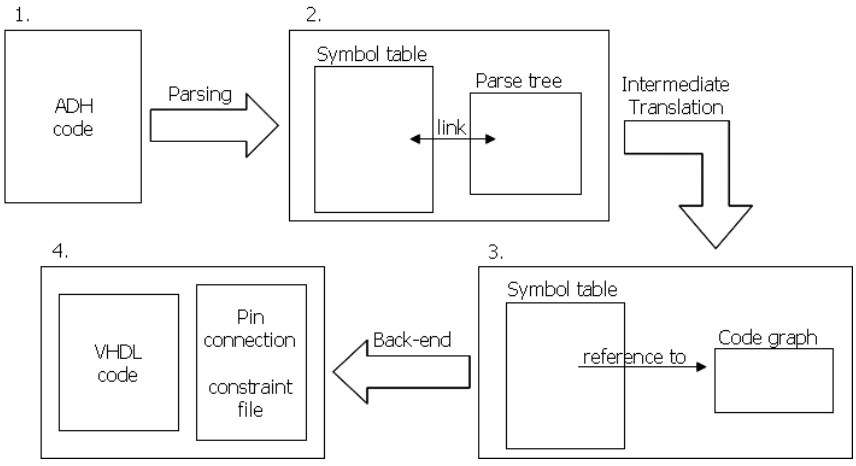


Figure 7.1: Overview of the compiling stages revised

and *method* construction statements (i.e. shell statements). The intermediate-stage concentrated on the statements within the *methods* (i.e. method statements). But now, the back-end uses all of the statements, including the shell statements and *method* statements, to generate the equivalent VHDL.

7.1 Back-end as VHDL

The back-end, starting from the control-flow graph (CFG), uses the following four step process.

1. Process the *class* from the CFG. This generates a *package* and *components*. The shell (defining the *package* and *component*) is created when it sees a node *class* and the inside is filled as it processes the *method*.

2. Process the *method* from the CFG, i.e. convert IR of the code-graph into structured/behavioural VHDL codes. This generates an entity and architecture for each *method*. Whether the type of VHDL is behavioural or structural depends on the contexts.

3. At the completion of 1. and 2. above, create the top-level architecture interconnecting the components generated by the *class* and *methods* (i.e. components instantiation).

4. Connect the pin-assignments. This generates a 'ucf' file.

The back-end starts processing from the top-node (called 'concerns') of a parse tree. It scans for a *method*, and then finds a code-graph to process. In order to understand the big picture of how the VHDL is produced, let us recall the structure of a class object, which is shown in Figure 7.2.

A package is created for a given *class*; refer to Figure 7.3. The package contains the components information, which can be obtained from the *method* arguments. And the *methods* in an object can be modelled as entities and architectures in VHDL. The *class* variables are to be accessed by other *methods*, and this is not obvious when modelled in VHDL as it has to be accessed by multiple entities. The

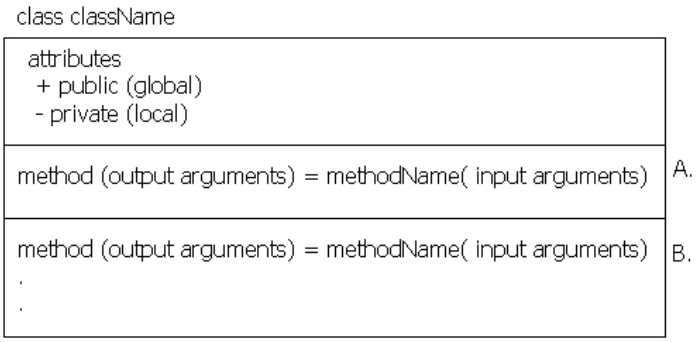


Figure 7.2: Structure of a typical class object

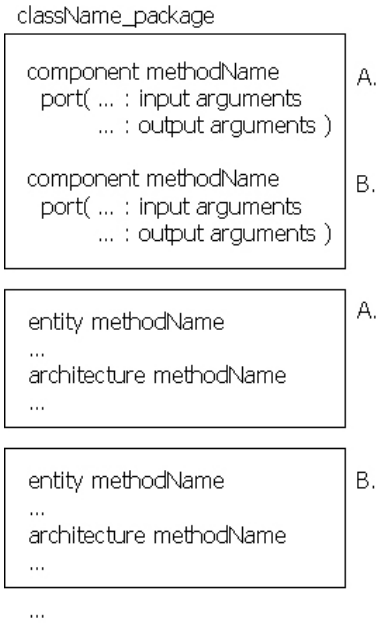


Figure 7.3: A class object representing a portion of equivalent VHDL files

variable should act as a separate storage (register). The detail of code generation is described in the following sub-sections.

7.2 Code generation

The major components in OO are the *class*, the *methods* and the *attributes* (class variables). Therefore, we present how each of these are processed to generate equivalent VHDLs in the following sub-sections.

7.2.1 Class

The *class* represents a group of *methods*. This grouping is described using a *package* in VHDL, and since the top-level will be using the class name itself, the package should have a different name. Hence, a package with naming convention, 'class-Name_package' is created when the compiler sees a class. It contains 'components', in which the number is the same as the number of *methods* in the *class*. The inputs and outputs are the same as the input and output arguments in a *method*. However, there is also one more additional implied input signal, which is a *method* call. Therefore, we have named the implied input as 'ENABLE' in an entity. It triggers the hardware block and processes the data in the hardware as if the function call is made in a microprocessor. For example,

```
class Simple {
   method(Integer result;) = Fred(Integer a; Integer b;) {
   ...
   }
}
```

will generate,

```
PACKAGE Simple_package IS
  COMPONENT Fred
    PORT (
      a : in Integer;
      b : in Integer;
      result : out Integer;
      ENABLE : in STD_LOGIC
    );
  END COMPONENT
END Simple_package
```

7.2.2 Method

A *method* creates a matching *entity*, which the input and output arguments determine as per the previous description. The naming convention is to use 'method-Name' as an entity name and 'methodName_arch' for the architecture name.

Each *method* has one more entry called 'execution token', which indicates that the *method* is being executed by hardware button press or called by another *method* or *class*. This is named as 'ENABLE' and it is included in the *entity*. It also requires

a signal at the top-level of VHDL to tell which *method* is being executed. Therefore, a signal name with a prefix 'EXEC_' followed by 'methodName' is also inserted at the top-level. This is further explained in the top-level generation sub-section.

The code-graph fills the architecture from the eight primitive statements. Six of them are directly converted into equivalent VHDL statements. The remaining two are the input and output pin connections that should be managed in the pin-connection process.

InstanceStatement

This requires an instantiation of a hardware component from a defined package. Some packages have been imported as a default, or some packages have been imported previously. In this case, a multiple importing is not required; hence, it can safely be discarded. Otherwise, a declaration statement is added. From the code-graph statement,

```
t@1 <- new Add()
```

the following is equivalent VHDL,

```
t@1 : Work.Add_package.add;
```

Note that this does not actually instantiate the hardware component yet. It is done when *CallStatement* is converted. The *InstanceStatement* is always followed by an *CallStatement*.

CallStatement

This requires an actual instantiation of a component and binding of the variables. Typical naming convention for the component is 'Uxxx' where xxx is an unique integer number incremented by 1 after each instantiation of an object. The equivalent VHDL statement is 'PORT MAP'. The usage of detailed 'PORT MAP' is described in VHDL Tutorial in Appendix B on page 117. For example,

```
t@2 <- t@1.add(a,b)
```

is translated to the equivalent VHDL,

```
U0 : t@1 PORT MAP (a, b, t@2);
```

The `t@1` is a defined component from the previous *InstanceStatement*.

EquateStatement

This is converted to a straight VHDL assignment. The VHDL statements are executed in parallel, and this is also stated in the ADH language description; hence, straight assignment using symbol '<=' can be done. For example, the assignment of a variable,

```
result <- t@2
```

yields an equivalent VHDL assignment,

```
result <= t@2
```

MultiplexerStatement

A multiplexer can be created using 'WITH ... SELECT' and 'WHEN' statements. Again this is described fully in the VHDL Tutorial. This uses the behavioural style of VHDL, which is better avoided if possible. For example,

```
t@5 <- Multiplexer(t@2, t@3, t@4 :: t@1)
```

generates the following VHDL statement,

```
WITH t@1 SELECT
  t@5 <= t@2 WHEN "00",
         t@3 WHEN "01",
         t@4 WHEN "10",
         'X' WHEN others;
```

ConditionalStatement

The *ConditionalStatement* is replaced by the *ResolutionStatement* by the process of SSA form generation. The *if*-like conditional statement is replaced by the *InstanceStatement* and *CallStatement*, and the dummy statement (for example, '`Test(t@4)`') is replaced by the *ResolutionStatement*.

The branches generated by the conditional control structure are both executed in parallel. The merging of the two branches and the choosing of which value to take from the branches are determined by the *ResolutionStatement*.

Consider the example in Figure 7.4. The conditional statement yields two branches as shown in the figure. For the microprocessors, only one branch is chosen

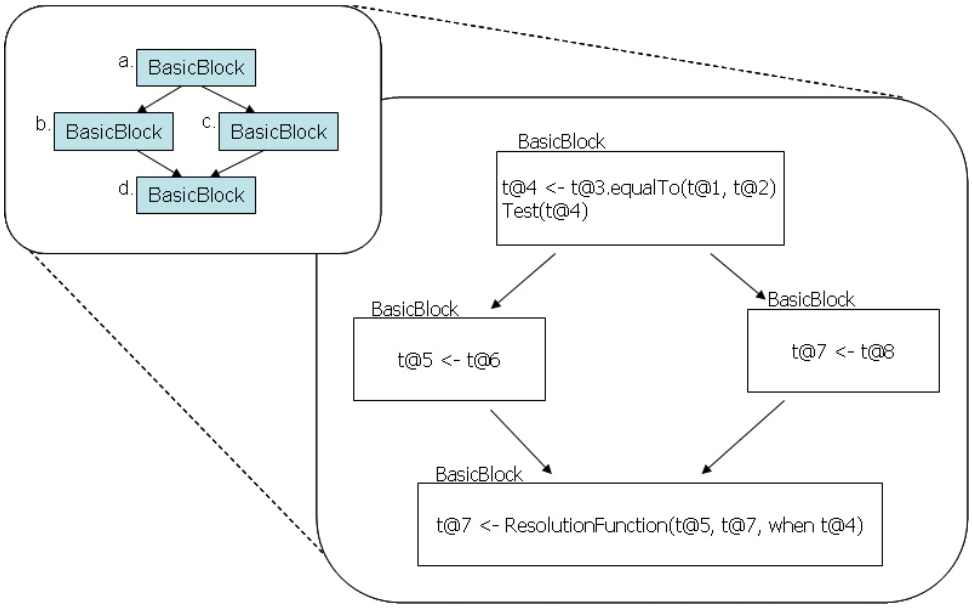


Figure 7.4: Basic blocks created by a conditional statement

and the dataflow path is determined from the top basic block ('a' basic block in the figure). However, for the FPGAs, both the branches ('b' and 'c' basic blocks) are executed and the phi-function, which chooses which value to take, is added at the merge of the two branches ('d' basic block).

ResulutionStatement

CASE and WHEN statements can be used to model the resolution statement. For example,

```
t@4 <- ResolutionFunction (t@1, t@2, when t@3)
```

is translated into equivalent VHDL,

```
WITH t@3 SELECT
   t@4 <= t@1 WHEN 0;
   t@4 <= t@2 WHEN others;
```

The correct logic is determined by the type of variable, `t@3`.

7.2.3 Class Variable

A *class* variable causes some difficulties in VHDL. In OO (Object-Oriented) programming languages, its functionality is a register that holds a value of a variable

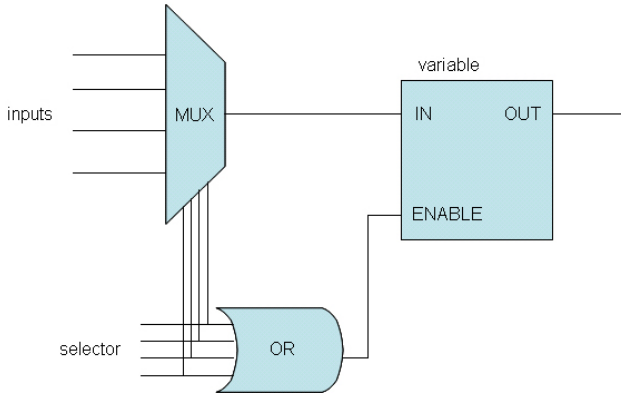


Figure 7.5: Class variable implementation

and the value is changed by other *methods*. Therefore, the class variable itself must become an entity and should be accessible to the other *methods* (functions/entity).

A *class* variable becomes a separate entity and architecture. Since the variable cannot be driven by the multiple drivers in the hardware, it also creates a multiplexer that has inputs that are the *methods* that access the variable. It creates a circuitry as shown in Figure 7.5.

The multiplexer is 'one-hot' coded (i.e. The binary code for the top input is 1000, and the next is 0100 and so on.). It does not have to be 'one-hot' coded, but for convenience, it has been implemented in this way at this stage. When a *method* accesses a variable, the logic level of the 'enable' pin is set at high and the value of a variable is passed to the multiplexer; therefore, it will be stored in the register. The output is the current register value and is always active.

7.2.4 Built-in method

The programming languages for microprocessors have a "main method" (or "main function") to locate where the starting point is in a program. In ADH, we also require a point where the execution starts. The *method* name 'process' in ADH is a reserved *method* name, which implements clocked circuitry using 'PROCESS' in VHDL. Therefore, the execution point can be given using a 'process' *method* that is driven by the hardware clock. As the power is supplied to the hardware, the master clock starts ticking and a 'process' *method* starts passing the 'execution token'.

'Process' itself is a keyword in VHDL; hence, it cannot be used as an entity

name. Therefore, another naming convention, 'processCLK', is used.

The following is an example of 'process' in ADH generating 'PROCESS' in VHDL. The detail of 'PROCESS' syntax in VHDL can be found in the VHDL tutorial in Appendix B.

```
method() = process(Integer a; Integer b;) {
...
```

generating the following,

```
ARCHITECTURE processCLK_arch of processCLK is
BEGIN
   PROCESS (clk)
      BEGIN
      IF (clk'EVENT AND clk = '1') THEN
      ...
```

Alternatively, the developer may require the process to happen at a certain frequency or in a certain time frame, for example, every clock cycle. It can be added into this 'process' *method* block.

Otherwise, the developer may force the statements to be executed in sequential order rather than in parallel. In this case, the statements can be written in the 'process' *method* block. Most of the circuitry is implemented in combinational logics to be implemented in parallel. However, in some cases, sequential order is required. For example,

```
a <- b;
c <- a;
```

It may seem apparent to the reader: the value of 'b' is assigned to 'a', which is passed to 'c'. However, recall that the statements in ADH are parallel. Then the value of 'b' is assigned to 'a' and at the same time, the value of 'a' is assigned to the 'c'. So it is not obvious that the value of 'c' is the old value of 'a' or the new value of 'a' assigned from 'b'. There is data dependency and this cannot be done in parallel. This must be implemented in a clocked circuitry using 'process' in VHDL.

Note that the given example is processed in sequential order anyway by the compiler after dataflow analysis. However, it is used to show that the developer can deliberately set statements to be executed in sequential order by writing the codes in a *method* called 'process'.

7.2.5 Top-level

We have shown how *class* objects can create separate VHDL equivalent codes. However, there is no connection between these components yet. This is done in the top-level of VHDL. The *class* and *methods* need to be combined in another entity and architecture as shown in Figure 7.6. In conventional HDL development, the top-level is often easier to implement in a schematic diagram rather than VHDL code, while the underneath components are written in VHDL codes. However, the top-level could still be implemented in structured VHDL and the process could be automated by the techniques described in this section.

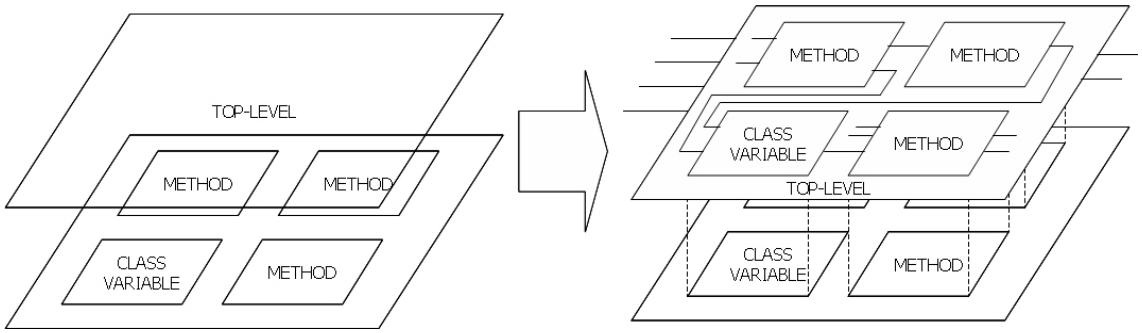


Figure 7.6: Overview of a VHDL

Each *method* is already created as a component in a package; therefore, it can easily be used again in the top-level. The top-level instantiates each *method* by calling the components. However, the connection between the class variable and the *methods* are difficult. The inputs and outputs of the entity need to be mapped to a common variable to be connected together as shown in Figure 7.7. Hence, another appropriate naming convention needs to be chosen.

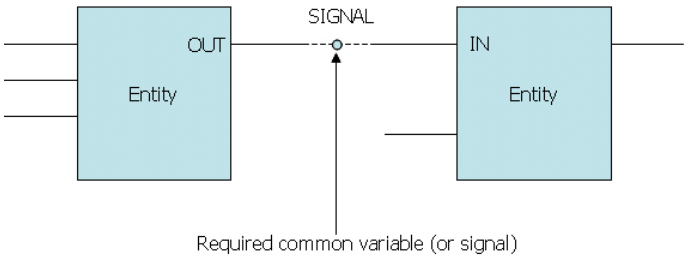


Figure 7.7: The required common signal when two entities are to be connected to each other

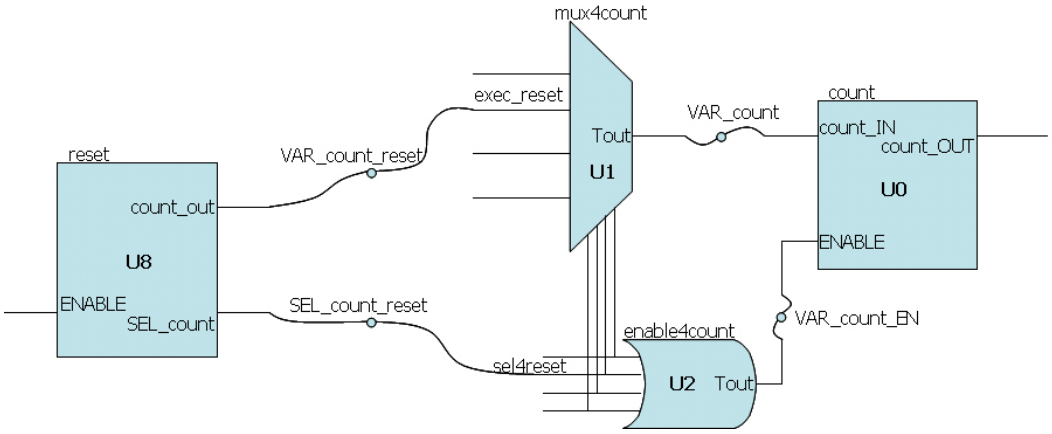


Figure 7.8: Partial circuit implementation of the top-level

For the inputs of the class variable, a prefix 'VAR_' and a suffix '_methodName' are used where the 'methodName' is actually the name of the *method*. For the selector pins of a multiplexer, a prefix 'SEL_' and a suffix '_methodName' are used where again 'methodName' is the actual name of the *method*.

Consider the following example, which is a code fragment that is fully shown in Chapter 8. The example contains a class variable and a *method* that accesses the variable. The class variable instantiate the circuitry described earlier and the *method* also instantiate a piece of hardware. They requires the connection between inputs and outputs.

```
public class Counter
{
  private Integer count <- 0;
  ...
  method () = reset() {
    count <- 0;
  }
  ...
```

Figure 7.8 shows an example of this top-level implementation. The figure consists of a class variable 'count' and a *method* 'reset', implemented as entities. When a 'reset' *method* is called (or executed), the implied input 'ENABLE' will be set to high, feeding the 'OUT' and 'SEL', which will result in changing the value of the count register.

The body of an architecture looks as follows:

```
ARCHITECTURE Counter_arch OF Counter IS
  VARIABLE VAR_count_reset: Integer;
  ...
  VARIABLE SEL_count_reset : STD_LOGIC;
  ...
BEGIN
  U1 : mux4count PORT MAP (
    EXEC_reset => VAR_count_reset,
    ...);
  U2 : enable4count PORT MAP (
    sel4reset => SEL_count_reset,
    ... );
  U8 : reset PORT MAP (
    count_OUT => VAR_count_reset,
    SEL_count => SEL_count_reset,
    ...);
  ...
END Counter_arch;
```

This creates an entire class into one re-useable package and an entity and architecture. The inputs to this entity are the sum of all the inputs to the *methods* depending on their visibility and also the implied inputs, which are the execution tokens we named as 'ENABLE'. And the outputs are the sum of all the *method* outputs depending on their visibility. Note that there are still some floating (missing) connection points such as the output of the 'count' variable. Those floating connection points are to be connected to the external pin constraint file and are connected to the hardware I/O pins.

7.2.6 Naming space

We have used the prefixing as the solution for the creation of matching variables for I/O. This may causes some naming clashes therefore we suggest an alternative solution apart from this naming convention solution. The use of a look-up table consisting of the input and output ports to be mapped. If a port is not inserted in the look-up table, adds it to the table and assign unique name, and if a port is already inserted in the look-up table, the unique name can be retrieved and it could be used to connect to the other port. This requires separate two-step processes using a symbol table. The first execution adds all the necessary ports into the

table and does some analysis, and second execution maps the ports. This has both advantages and disadvantages. An advantage is that the compiler becomes more robust by adding some more analysis and preventing the name clashes; however, it requires multi-path process that takes time and also could provide places for some bugs to appear. Therefore, the usefulness can be debated. In the end, the naming convention solution is faster since the top-level VHDL can be written in only a one step process whereas the look-up table takes some time every time adding and looking up the table.

7.3 Pin connection

The pin connection has not been fully implemented; however, it could easily be implemented by examining the 'ucf' files generated by a vender specific tool. The syntax is very simple and as follows:

```
NET name LOC = "Pxxx" ;
```

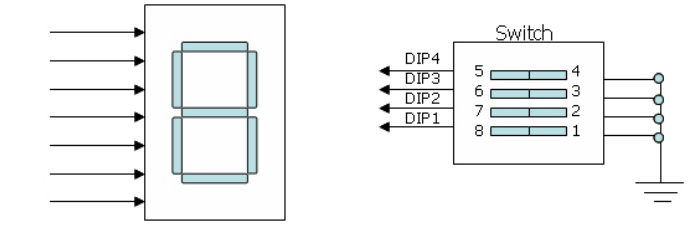
The keyword 'NET' specifies that it is a pin connection statement. It is followed by the name of the input/output port in the entity. The keyword 'LOC' tells the location of the pin where Pxxx is the specified pin number on the physical hardware device. The line finishes with a semicolon as usual, and there could be one more optional argument that we could set the default value of the pin according to whether it is 'PULLUP' or 'PULLDOWN'.

Figure 7.9 shows a code extract from the Xilinx PACE tool, and it simply describes 'clk' being connected to pin number 185 and 'clr' connected to pin number 16 with default being 'PULLUP'. The rest are the connections to the seven-segment LED and the ON/OFF control pin for the LED for the particular example.

The default 'PULLUP' or 'PULLDOWN' must also be supplied by a developer as they are determined by a specific hardware device. Figure 7.10 shows two examples of hardware devices. Figure 7.10 (a) shows a seven-segment LED driven by a hardware device and 7.10 (b) shows a DIP switch with the four pins connected to the hardware board and the other four connected to the ground. In this case (Figure 7.10 (b)), an internal pull-up is required to generate a logic high.

```
NET "clk"  LOC = "P185"  ;
NET "clr"  LOC = "P16" | PULLUP ;
NET "count<0>"  LOC = "P18"  ;
NET "count<1>"  LOC = "P60"  ;
NET "count<2>"  LOC = "P59"  ;
NET "count<3>"  LOC = "P57"  ;
NET "count<4>"  LOC = "P58"  ;
NET "count<5>"  LOC = "P63"  ;
NET "count<6>"  LOC = "P62"  ;
NET "count<7>"  LOC = "P61"  ;
```

Figure 7.9: Pin connection code extract from Xilinx PACE tool.



(a) 7-Segment LED Display Interface (b) 4-Pin DIP Switch Interface

Figure 7.10: Pinout descriptions of hardware devices

7.4 Summary

The back-end generation that produces VHDL output has been described in this chapter. The pin-connection has not been fully implemented; therefore, a suggested *method* is given instead. The last three chapters have shown the necessary steps in implementing this compiler in detail. The next chapter shows the result of what this compiler produces at this stage and what is the current state of the compiler by describing some of the known bugs.

Chapter 8

Result

In this chapter, the current state of the compiler is presented with the current outputs that the compiler generates. There are a number of issues to be solved in order to actually produce a fully working hardware device code, which are presented in sub-section 8.2.

8.1 Result

To see how the compiler performs and whether it produces the designed and useable result, we have designed a simple 'counter example' that could be implemented on the Xilinx Spartan-II platform. The basic concept is like a stopwatch, displaying a count number and three buttons to start, to stop and to reset the counter as shown in Figure 8.1.

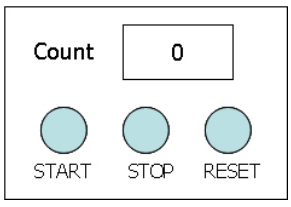


Figure 8.1: Conceptual view of the counter example

From the software programmer's view, this can simply be done via the 'state machine' concept.

```
Start -> Counting -> Stop
```

It requires a variable to store the current value of the counter, and a variable to store the current state of the counter, and the buttons can be written in corresponding methods as shown in Figure 8.2.

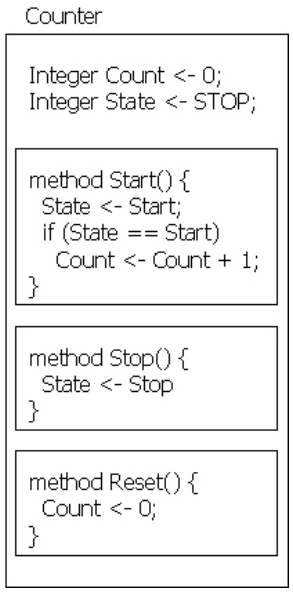


Figure 8.2: Software design for a counter example

Figure 8.3 shows an actual counter example written in ADH. It is written with two *class* variables and four *methods* to see the interactions between the multiple *class* variables and the multiple *methods* accessing the variables. It could be written using the FSM control structure that we have designed in ADH; however, it is worthwhile investigating the standard programming structure for microprocessors. One of the aims of this project is to utilise the highly trained software engineers (as well as signal and image processing engineers who have experience in programming languages, such as MATLAB) to develop and configure the FPGAs.

After compilation through the compiler, Figures 8.6 – 8.13 on pages 100 – 105 are produced. It may seem that the generated VHDL listings are more than adequate. Obviously, a lot less VHDL code would be written by hardware engineers in terms of the LoC (Line of Code). However, humans program in the higher-level description and the job of the VHDL compiler is to produce the equivalent netlist. In this case, we have produced a lot of structural styles of VHDL that are closer to the netlist compared to the behavioural style that is what a programmer would write. Also

```
1:  public class Counter
2:  {
3:    private Integer count <- 0;
4:    private Integer states <- 0; // 0 for stop, 1 for start.
5:
6:    method () = start() {
7:      states <- 1;
8:    }
9:
10:   method () = process() {
11:     when (states == 1) {
12:       count <- count + 1;
13:     }
14:     connection("Pin21") <- count;
15:   }
16:
17:   method () = reset() {
18:     states <- 0;
19:     count <- 0;
20:   }
21:
22:   method () = stop() {
23:     states <- 1;
24:   }
25:
26: }
```

Figure 8.3: A counter example in ADH

we have defined and used our own library supporting functions rather than use the VHDL built-in functions. Therefore, our compiler is not as inefficient as it may seem because the hardware resource usage would not be proportional to the LoC.

The effective schematic diagram is shown in Figure 8.4 as a reference. Note that the labelling of the entities in VHDL and the schematic diagram match each other but not all the I/O ports are labelled. This is for illustration purposes only. One more thing to note is that the input pin connections are not yet defined in the ADH. Pressing the hardware buttons should call associated *methods* in ADH with the pin connection command.

There is one more example code that has been designed but unable to produce output at this stage due to some bugs in the compiler. Figure 8.5 is a DTMF (Dual-Tone Multi-Frequency) tone decoder written in ADH. The earlier example

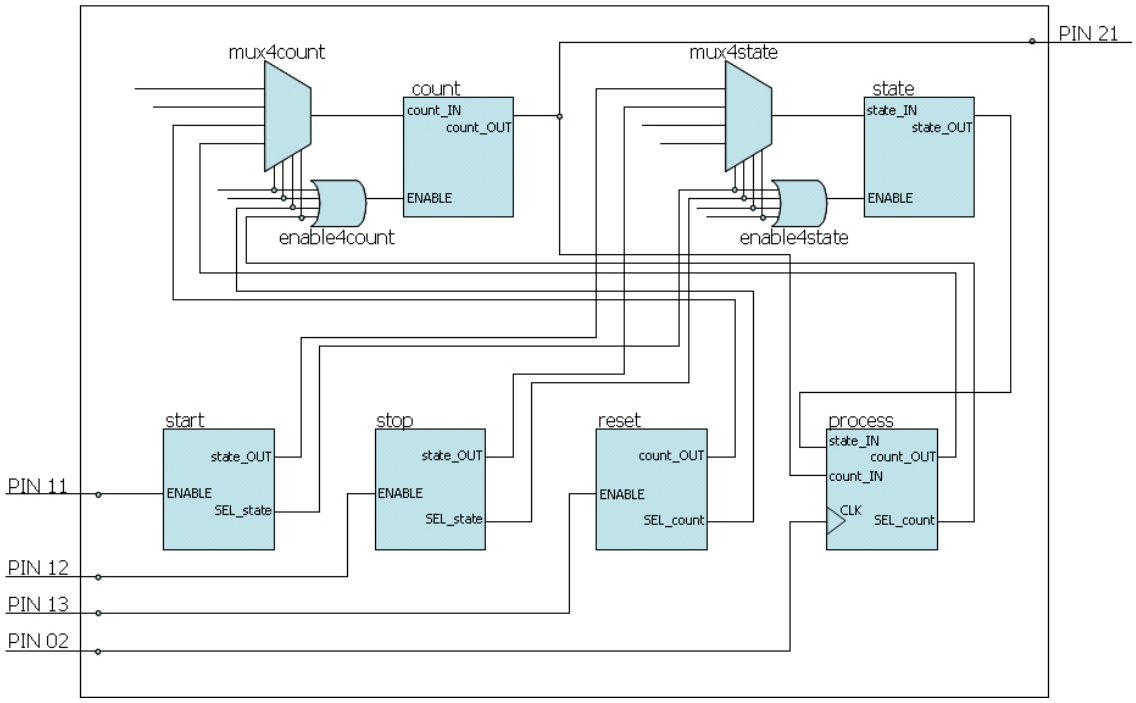


Figure 8.4: Equivalent schematic of the counter example

was the control modelling problem and this is a problem applicable to the signal and image processing domains. We add this example, even though the compiler cannot produce reasonable output at this stage due to some bugs, because it can still show the powerfulness and the usefulness of this new language, ADH.

```
 1: public class DTMFdecoder {
 2:     private Integer fundamentalCoefficients[][] <- {{},{},{},{},{},{},{}}; // coefficients to be determined
 3:     private Integer harmonicCoefficients[][] <- {{},{},{},{},{},{},{}};
 4:     private Object toneCodePattern[][]
 5:         <- {{"1", 0b1000100}, {"2", 0b1000010}, {"3", 0b1000001}, {"4", 0b0100100},
 6:             {"5", 0b0100010}, {"6", 0b0100001}, {"7", 0b0010100}, {"8", 0b0010010},
 7:             {"9", 0b0010001}, {"0", 0b0001100}, {"*", 0b0001010}, {"#", 0b0001001}};
 8:
 9:     public method () = process() {
10:         PCMinput tone = new PCMinput(8000); // incoming signals are stored in this buffer
11:         String output = decode(tone);
12:     }
13:
14:     private method (String result) = decode(PCMinput tone) {
15:         Object output[7];
16:         output <- detector(tone);
17:         result <- decisionLogic(output);
18:     }
19:
20:     private method (Integer tones[]) = detector(PCMinput input) {
21:         SecondOrderIIR fundamentalFilters[] <- fundamentalCoefficients.map(f, new SecondOrderIIR(f));
22:         SecondOrderIIR harmonicFilters[] <- harmonicCoefficients.map(f, new SecondOrderIIR(f));
23:         Integer fundamentals[] <- fundamentalFilters.filter(input);
24:         Integer harmonics[] <- harmonicFilters.filter(input);
25:         tones <- fundamentals .^ 2 + harmonics .^ 2;
26:     }
27:
28:     private method (String decodedTone;) = decisionLogic(Integer tones[];) {
29:         Boolean toneCode[] <- tones > THRESHOLD;
30:         for (i <- 0..11) {
31:             decodedTone <- toneCodePattern[i][0] when (toneCode == (toneCodePattern[i][1]));
32:         }
33:     }
34: }
35:
36: public class SecondOrderIIR {
37:     private Integer filterCoefficients[];
38:
39:     public SecondOrderIIR(Integer coeffients[];) {
40:         filterCoefficients = coefficients;
41:     }
42:
43:     public method (Integer output;) = filter(Integer input;) {
44:         Integer w0, w1, w2;
45:         w0 <- input + w1 * filterCoefficients[0] + w2 * filterCoefficients[1];
46:         w1 <- w0; w2 <- w1;
47:         output <- w0 + w1 * filterCoefficients[2];
48:     }
49: }
50: ------------------------------------------------------------
51: public aspect Debug {
52:
53:     LCDdisplay disp = new LCDdisplay(2, 16);
54:
55:     pointcut decisions(Integer toneCode)
56:         : call (DTMFdecoder.decisionLogic(Integer)) && args(toneCode);
57:
58:     before(Integer abc) : decisions(abc) {
59:         disp.show("ToneCode: " + abc);
60:     }
61: }
```

Figure 8.5: DTMF tone decoder written in ADH

8.2 Current compiler & Known bugs

There is some more work to be done in order to actually produce a working hardware device. Some of the known bugs and possible solutions are suggested in this section.

First, the generated VHDL is still not synthesizable due to missing library classes. The major datatypes such as *Integer* and *String* are not implemented yet; major decisions on the bit representation are yet to be made. The first step would be to use a fixed length for every value. Say, 8-bit *Integer* with `STD_LOGIC_VECTOR(7 downto 0)`. There is also the IEEE 1164 standard library [14] that provides the conversion between the datatypes.

Second, simple operators such as arithmetic operators and comparators (<, >, ==, etc.) also need to be implemented as we treat all of them as functions. They can be implemented either as library classes or built-in functions that use VHDL operators directly. The first step would be to use VHDL operators directly; however, the future plan is to generate the netlist directly. In this case, using VHDL operators would not work and implementation of library classes is required.

Third, there are currently some bugs in the code including the SSA form not correctly producing its result, making some of the statements disappear in optimisation (dead-code elimination). The task of the dead-code elimination is to eliminate some of the unreachable codes, but some of the useful statements also disappear at this stage. This bug does not directly affect the result; hence, the optimisation can be safely turned off to prevent mis-behaviours.

Fourth, some of the language features such as the *set* do not operate at the moment. It is similar to *enum* in the latest *Java5* as described earlier in Chapter 3. In the 'counter example', the following *set* was written.

```
public set State {
  START,
  STOP
}
```

And in the variable assignment code for the states, the following was the original coding,

```
private Integer states <- START;
```

However, there are some bugs in the code in that it does not correctly recognise the names in a *set*. Therefore, this also needs to be looked at. Some of the bugs are minor in the sense that there are alternative programming solutions to bypass these errors.

Fifth, more work also needs to be done on the top-level VHDL production such as mapping of the I/O ports. There is also a bug that places some of the variable definitions within the body of the architecture rather than within the header of the architecture. This is just a software design and implementation issue. Currently, once the outputs for the architecture body have been started, the outputs to the architecture header are not possible because they are written into the same buffer. A possible solution is to use multiple buffers and multiple pointers to generate the outputs. This will allow the compiler to be able to access the architecture body and the architecture header as well as the entities and the package. Therefore a better design for the back-end output needs to be carefully considered.

In this chapter, we have shown the current outputs that the compiler produces and also have described the current known bugs and possible solutions to them.

```
 1: USE Work.Add_package.Add;
 2: USE Work.EqualTo_package.EqualTo;
 3: USE Work.Integer_package.Integer;
 4:
 5: PACKAGE Counter_package IS
 6:   COMPONENT count
 7:     PORT (
 8:       count_IN : in Integer;
 9:       count_OUT : out Integer;
10:       ENABLE : in BIT
11:     );
12:   END COMPONENT
13:   COMPONENT mux4count
14:     PORT (
15:       SEL : in STD_LOGIC_VECTOR (3 DOWNTO 0);
16:       EXEC_start : in Integer;
17:       EXEC_stop : in Integer;
18:       EXEC_reset : in Integer;
19:       EXEC_processCLK : in Integer;
20:       Tout : out Integer
21:     );
22:   END COMPONENT
23:   COMPONENT enable4count
24:     PORT (
25:       SEL_start : in STD_LOGIC;
26:       SEL_processCLK : in STD_LOGIC;
27:       SEL_reset : in STD_LOGIC;
28:       SEL_stop : in STD_LOGIC;
29:       Tout : out STD_LOGIC
30:     );
31:   END COMPONENT
32:   COMPONENT state
33:     PORT (
34:       states_in : in Integer;
35:       states_out : out Integer;
36:       ENABLE : in BIT
37:     );
38:   END COMPONENT
39:   COMPONENT mux4state
40:     PORT (
41:       SEL : in STD_LOGIC_VECTOR (3 DOWNTO 0);
42:       EXEC_start : in Integer;
43:       EXEC_stop : in Integer;
44:       EXEC_reset : in Integer;
45:       EXEC_processCLK : in Integer;
46:       Tout : out Integer
47:     );
48:   END COMPONENT
49:   COMPONENT enable4state
50:     PORT (
51:       SEL_start : in STD_LOGIC;
52:       SEL_stop : in STD_LOGIC;
53:       SEL_reset : in STD_LOGIC;
54:       SEL_processCLK : in STD_LOGIC;
55:       Tout : out STD_LOGIC
56:     );
57:   END COMPONENT
58:   COMPONENT start
59:     PORT (
60:   VAR_count_OUT : out Integer;
61:       SEL_count : out STD_LOGIC;
62:       ENABLE : in STD_LOGIC
63:     );
64:   END COMPONENT
65:   COMPONENT stop
66:     PORT (
67:       VAR_state_OUT : out Integer;
68:       SEL_state : out STD_LOGIC;
69:       ENABLE : in STD_LOGIC
70:     );
71:   END COMPONENT
72:   COMPONENT reset
73:     PORT (
74:       VAR_count_OUT : out Integer;
75:       SEL_count : out STD_LOGIC;
76:       ENABLE : in STD_LOGIC
77:     );
78:   END COMPONENT
79:   COMPONENT processCLK
80:     PORT (
81:       VAR_count_IN : in Integer;
82:       VAR_state_IN : in Integer;
83:       VAR_count_OUT : out Integer;
84:       SEL_count : out STD_LOGIC;
85:       CLK : in STD_LOGIC
86:     );
87:   END COMPONENT
88: END Counter_package
```

Figure 8.6: The generated VHDL of the package for ‘counter example’

```
 1:  ENTITY Counter IS
 2:    PORT (
 3:      EXEC_start : in STD_LOGIC;
 4:      EXEC_stop : in STD_LOGIC;
 5:      EXEC_reset : in STD_LOGIC;
 6:      EXEC_processCLK : in STD_LOGIC;
 7:      VAR_count_OUT : out STD_LOGIC
 8:    );
 9:
10:  ARCHITECTURE Counter_arch OF Counter IS
11:    VARIABLE VAR_count_IN : Integer;
12:    VARIABLE VAR_count_OUT : Integer;
13:    VARIABLE VAR_count_start : Integer;
14:    VARIABLE VAR_count_stop : Integer;
15:    VARIABLE VAR_count_reset : Integer;
16:    VARIABLE VAR_count_processCLK : Integer;
17:    VARIABLE SEL_count_IN : STD_LOGIC;
18:    VARIABLE SEL_count_start : STD_LOGIC;
19:    VARIABLE SEL_count_stop : STD_LOGIC;
20:    VARIABLE SEL_count_reset : STD_LOGIC;
21:    VARIABLE SEL_count_processCLK : STD_LOGIC;
22:
23:    VARIABLE VAR_state_IN : Integer;
24:    VARIABLE VAR_state_OUT : Integer;
25:    VARIABLE VAR_state_start : Integer;
26:    VARIABLE VAR_state_stop : Integer;
27:    VARIABLE VAR_state_reset : Integer;
28:    VARIABLE VAR_state_processCLK : Integer;
29:    VARIABLE SEL_state_IN : STD_LOGIC;
30:    VARIABLE SEL_state_start : STD_LOGIC;
31:    VARIABLE SEL_state_stop : STD_LOGIC;
32:    VARIABLE SEL_state_reset : STD_LOGIC;
33:    VARIABLE SEL_state_processCLK : STD_LOGIC;
34:
35:  BEGIN
36:    U0 : count PORT MAP (
37:      count_IN => VAR_count_IN,
38:      count_OUT => VAR_count_OUT,
39:      ENABLE =>  SEL_count_IN
40:    );
41:    U1 : mux4count PORT MAP (
42:      SEL => (SEL_count_start & SEL_count_stop &
SEL_count_reset & SEL_count_processCLK),
43:      EXEC_start => VAR_count_start,
44:      EXEC_stop => VAR_count_stop,
45:      EXEC_reset => VAR_count_reset,
46:      EXEC_processCLK => VAR_count_processCLK
47:    );
48:    U2 : enable4count PORT MAP (
49:      SEL_start => SEL_count_start,
50:      SEL_stop => SEL_count_stop,
51:      SEL_reset => SEL_count_reset,
52:      SEL_processCLK => SEL_count_processCLK,
53:      Tout =>  SEL_count_IN
54:    );
55:    U3 : state PORT MAP (
56:      state_IN => VAR_state_IN,
57:      state_OUT => VAR_state_OUT,
58:      ENABLE => SEL_state_IN
59:    );
61:    U4 : mux4state PORT MAP (
62:      SEL => (SEL_state_start & SEL_state_stop &
SEL_state_reset & SEL_state_processCLK),
63:      EXEC_start => VAR_state_start,
64:      EXEC_stop => VAR_state_stop,
65:      EXEC_reset => VAR_state_reset,
66:      EXEC_processCLK => VAR_state_processCLK
67:    );
68:    U5 : enable4count PORT MAP (
69:      SEL_start => SEL_state_start,
70:      SEL_stop => SEL_state_stop,
71:      SEL_reset => SEL_state_reset,
72:      SEL_processCLK => SEL_state_processCLK,
73:      Tout =>  SEL_state_IN
74:    );
75:    U6 : start PORT MAP (
76:      ENABLE => EXEC_start,
77:      states_OUT => VAR_state_start,
78:      SEL_start => SEL_state_start
79:    );
80:    U7 : stop PORT MAP (
81:      ENABLE => EXEC_stop,
82:      states_OUT => VAR_state_stop,
83:      SEL_stop => SEL_state_stop
84:    );
85:    U8 : reset PORT MAP (
86:      ENABLE => EXEC_reset,
87:      count_OUT => VAR_count_reset,
88:      SEL_reset => SEL_count_reset
89:    );
90:    U9 : processCLK PORT MAP (
91:      CLK => EXEC_processCLK,
92:      count_IN => VAR_count_OUT,
93:      state_IN => VAR_state_OUT,
94:      count_OUT => VAR_count_processCLK,
95:      SEL_processCLK => SEL_count_processCLK
96:    );
97:  END Counter_arch
```

Figure 8.7: The generated VHDL of the top-level for ‘counter example’

```
 1:  ENTITY count IS
 2:    PORT (
 3:      count_in : in Integer;
 4:      count_out : out Integer;
 5:      ENABLE : in STD_LOGIC
 6:    );
 7:  END count
 8:
 9:  ARCHITECTURE count_arch OF count IS
10:    SIGNAL temp : Integer;
11:  BEGIN
12:    temp <= count_in WHEN (ENABLE == 1);
13:    count_out <= temp;
14:  END count_arch;
15:
16:  ENTITY mux4count
17:    PORT (
18:      SEL : in STD_LOGIC_VECTOR (5 DOWNTO 0);
19:      exec_start : in Integer;
20:      exec_process : in Integer;
21:      exec_reset : in Integer;
22:      exec_stop : in Integer;
23:      exec_getCount : in Integer;
24:      Tout : out Integer
25:    );
26:  END mux4count
27:
28:  ARCHITECTURE mux4count_arch OF mux4count IS
29:  BEGIN
30:    PROCESS (SEL)
31:    BEGIN
32:      CASE SEL IS
33:        WHEN "00001" => Tout <= exec_start;
34:        WHEN "00010" => Tout <= exec_process;
35:        WHEN "00100" => Tout <= exec_reset;
36:        WHEN "01000" => Tout <= exec_stop;
37:        WHEN "10000" => Tout <= exec_getCount;
38:      END CASE;
39:    END PROCESS;
40:  END mux4count_arch;
41:
42:  ENTITY enable4count
43:    PORT (
44:      sel4start : in STD_LOGIC;
45:      sel4process : in STD_LOGIC;
46:      sel4reset : in STD_LOGIC;
47:      sel4stop : in STD_LOGIC;
48:      sel4getCount : in STD_LOGIC;
49:      Tout : out STD_LOGIC
50:    );
51:  END enable4count;
52:
53:  ARCHITECTURE enable4count_arch OF enable4count IS
54:  BEGIN
55:    Tout <= sel4start or sel4process or sel4reset or sel4stop or sel4getCount;
56:  END enable4count_arch;
```

Figure 8.8: The generated VHDL of the class variable ‘count’ for ‘counter example’

```
 1:  ENTITY states IS
 2:    PORT (
 3:      states_in : in Integer;
 4:      states_out : out Integer;
 5:      ENABLE : in STD_LOGIC
 6:    );
 7:  END states
 8:
 9:  ARCHITECTURE states_arch OF states IS
10:    SIGNAL temp : Integer;
11:  BEGIN
12:    temp <= states_in WHEN (ENABLE == 1);
13:    states_out <= temp;
14:  END states_arch;
15:
16:  ENTITY mux4states
17:    PORT (
18:      SEL : in STD_LOGIC_VECTOR (5 DOWNTO 0);
19:      exec_start : in Integer;
20:      exec_process : in Integer;
21:      exec_reset : in Integer;
22:      exec_stop : in Integer;
23:      exec_getCount : in Integer;
24:      Tout : out Integer
25:    );
26:  END mux4states
27:
28:  ARCHITECTURE mux4states_arch OF mux4states IS
29:  BEGIN
30:    PROCESS (SEL)
31:    BEGIN
32:      CASE SEL IS
33:        WHEN "00001" => Tout <= exec_start;
34:        WHEN "00010" => Tout <= exec_process;
35:        WHEN "00100" => Tout <= exec_reset;
36:        WHEN "01000" => Tout <= exec_stop;
37:        WHEN "10000" => Tout <= exec_getCount;
38:      END CASE;
39:    END PROCESS;
40:  END mux4states_arch;
41:
42:  ENTITY enable4states
43:    PORT (
44:      sel4start : in STD_LOGIC;
45:      sel4process : in STD_LOGIC;
46:      sel4reset : in STD_LOGIC;
47:      sel4stop : in STD_LOGIC;
48:      sel4getCount : in STD_LOGIC;
49:      Tout : out STD_LOGIC
50:    );
51:  END enable4states;
52:
53:  ARCHITECTURE enable4states_arch OF enable4states IS
54:  BEGIN
55:    Tout <= sel4start or sel4process or sel4reset or sel4stop or sel4getCount;
56:  END enable4states_arch;
```

Figure 8.9: The generated VHDL of the class variable ‘state’ for ‘counter example’

```
1:   ENTITY start is
2:     PORT (
3:       states_OUT : out Integer;
4:       SEL_start : out STD_LOGIC;
5:       ENABLE : in STD_LOGIC
6:     );
7:   END start;
8:
9:   ARCHITECTURE start_arch of start is
10:  BEGIN
11:    states_OUT <= 1 WHEN (ENABLE == 1);
12:    SEL_start <= 1 WHEN (ENABLE == 1);
13:  END start_arch;
```

Figure 8.10: The generated VHDL of the method ‘start’ for ‘counter example’

```
1:   ENTITY stop is
2:     PORT (
3:       states : out Integer;
4:       SEL_states : out STD_LOGIC;
5:       ENABLE : in STD_LOGIC
6:     );
7:   END stop;
8:
9:   ARCHITECTURE stop_arch of stop is
10:  BEGIN
11:    states <= 1 when (ENABLE == 1);
12:    SEL_states <= 1 when (ENABLE == 1);
13:  END stop_arch;
```

Figure 8.11: The generated VHDL of the method ‘stop’ for ‘counter example’

```
1:   ENTITY reset is
2:     PORT (
3:       states : out Integer;
4:       SEL_states : out STD_LOGIC;
5:       count : out Integer;
6:       SEL_count : out STD_LOGIC;
7:       ENABLE : in STD_LOGIC
8:     );
9:   END reset;
10:
11:  ARCHITECTURE reset_arch of reset is
12:  BEGIN
13:    states <= 0 when (ENABLE == 1);
14:    SEL_states <= 1 when (ENABLE == 1);
15:    count <= 0 when (ENABLE == 1);
16:    SEL_count <= 1 when (ENABLE == 1);
17:  END reset_arch;
```

Figure 8.12: The generated VHDL of the method ‘reset’ for ‘counter example’

```
 1:   ENTITY process is
 2:     PORT (
 3:       count_OUT : out Integer;
 4:       count_IN : in Integer;
 5:       SEL_processCLK : out STD_LOGIC;
 6:       CLK : in STD_LOGIC
 7:     );
 8:   END process;
 9:
10:   ARCHITECTURE process_arch of process is
11:     SIGNAL t@2 : Integer;
12:     SIGNAL t@4 : Integer;
13:   BEGIN
14:     PROCESS (clk)
15:       BEGIN
16:       IF (clk'EVENT AND clk = '1') THEN
17:         t@3 : Work.EqualTo_package.equalTo;
18:         U0 : t@3 PORT MAP (1, 1, t@4);
19:         t@1 : Work.Add_package.add;
20:         U1 : t@1 PORT MAP (count_IN, 1, t@2);
21:         count_OUT <= t@2 WHEN (t@4 == TRUE);
22:         SEL_processCLK <= 1;
23:       END IF;
24:     END PROCESS;
25:   END process_arch;
```

Figure 8.13: The generated VHDL of the method ‘process’ for ‘counter example’

Chapter 9

Conclusion

9.1 Discussion and conclusion

This project involved the design of a new hardware programming language and an associated compiler that produces executable code on FPGAs; the language is targeted for the signal and image processing domain. A new programming language is desirable for this application for a number of reasons. Because FPGAs provide high bandwidth and high datarate processing power, they are able to solve many signal processing problems that microprocessors (especially DSPs) are unable to perform efficiently. However, the lack of available programming tools has made FPGAs difficult to use. The recent software programming technique, aspect-oriented software development (AOSD), has been looked at and some of its useful techniques have been brought into the hardware programming language.

We have successfully designed a new language, ADH, which supports aspect-oriented features. It also has some new features such as a FSM control structure, which is especially useful for the specific application domain. At this stage, the compiler is able to produce synthesizable VHDL language. Although some of the features are still incomplete and currently the generated VHDL cannot directly be used on the hardware, small sections of the generated VHDL can be synthesized and with minor changes to the code, it can be executed on the hardware.

The major tasks are the completion of basic functionality of ADH language features and its compiler. Even though there is some more work that needs to

be done to increase the stability and the usability of this compiler, the result was promising; therefore, more research could be carried out in this direction.

Static aspect features, static introduction and static cross-cutting can be woven as planned. Hence, the first move to the AOP is implemented successfully. The next step is to move onto the dynamic cross-cutting and work out better ways to implement weaving.

To discuss the effectiveness of the language, a simple method of counting the line-of-code is used. Although the line-of-code does not tell how effective our language is, we have reduced the coding lines significantly. And moving from the structure based language to object-oriented and aspect-oriented language is also a significant improvement.

This language and the compiler would help in developing hardware design and implementation, in particular the signal and image processing domain problems, by natural OO design that can be used with ADH and avoids the need to understand the digital-logic implementation in detail.

9.2 Future work

This project has provided a framework for future possible ADH projects to extend the functionality of the language and also to tackle the current bugs. It also provides research areas to do with the other related issues.

The clocking and the timing analysis is one aspect that we have not considered in this thesis. The timing is one of the major issue in digital hardware and a programmer might want to get hold of the clock signal and do something with it. It should be included in the language syntax to explicitly capture the master clock so a programmer can use it appropriately.

The automated VHDL production might waste some of the hardware resources (number of logics used) compared to human written VHDL. A possible solution to this could be found through optimisation techniques or directly generating netlists without producing the VHDL codes.

The automatic type conversions, allowing the change of the variable length such

as *Integer* and *Real* by the compiler, and their affect on the SNR (Signal-to-Noise Ratio) could also be investigated.

Finally, the dynamic aspect features based on intermediate-stage code-graph modification could provide more powerful weaving.

In summary, the future work areas are given as follow:

- Tackling the bugs in the current compiler to increase the stability and usability
- Implementation of the library functions
- Refactoring and re-designing some of the classes
- Performance evaluation
- Optimisation techniques
- Dynamic aspect feature implementation
- Variable Integer-width and SNR
- Bypassing the VHDL generation and directly generating the netlist

The listings are given in order of importance. As the compiler is still not in 100% working order, the remaining bugs should be tackled first followed by either the implementation of the library functions or the refactoring and re-designing some of the components. Some re-design proposals have been described throughout the compiler implementation chapters, Chapter 5, 6, and 7. When this is done, the compiler should be able to produce synthesizable hardware devices, which is where the performance evaluation could be carried out.

Appendix A

BNF of ADH

```
NON-TERMINALS

TranslationUnit ::= ( ClassDefinition | AspectDefinition |
                    SetDefinition )* EndOfFile
EndOfFile ::= ( "." | <EOF> )
ClassDefinition ::= ( VisibilityDefinition <CLASS> Identifier
                    <OPENBRACE> ( AttributeDeclaration |
                    ConstructorDeclaration | MethodDeclaration )*
                    <CLOSEBRACE> )
AspectDefinition ::= ( VisibilityDefinition <ASPECT> Identifier
                     <OPENBRACE> ( StaticIntroduction |
                     AttributeDeclaration | MethodDeclaration |
                     PointcutDeclaration | AdviceDeclaration )*
                     <CLOSEBRACE> )
SetDefinition ::= ( VisibilityDefinition <SET> Identifier <OPENBRACE>
                  ( Identifier <SEMICOLON> )* <CLOSEBRACE> )
VisibilityDefinition ::= ( <VISIBILITY> )?
AttributeDeclaration ::= ( VisibilityDefinition EntityType Identifier
                         EntityDimensions ( <ASSIGN> Initializer )?
                         ( <COMMA> Identifier EntityDimensions
                         ( <ASSIGN> Initializer )? )* <SEMICOLON> )
```

```
EntityType ::= ( <IDENTIFIER> )
EntityDimensions ::= ( <OPENBRACKET> Expression <CLOSEBRACKET> )*
Initializer ::= ( Expression | <OPENBRACE> ( Expression ( <COMMA>
                Expression )* )? <CLOSEBRACE> )
ConstructorDeclaration ::= ( VisibilityDefinition Identifier
                           InputArguments StatementBlock )
MethodDeclaration ::= ( VisibilityDefinition <METHOD> OutputArguments
                      <EQUAL> Identifier InputArguments
                      StatementBlock )
OutputArguments ::= <OPENPARENTHESIS> ( AttributeDeclaration )*
                    <CLOSEPARENTHESIS>
InputArguments ::= <OPENPARENTHESIS> ( AttributeDeclaration )*
                   <CLOSEPARENTHESIS>
PointcutDeclaration ::= ( VisibilityDefinition <POINTCUT> Identifier
                        InputArguments <COLON> <CALL>
                        <OPENPARENTHESIS> <METHOD> OutputArguments
                        <EQUAL> MethodReference InputArguments
                        <CLOSEPARENTHESIS> <SEMICOLON> )
AdviceDeclaration ::= ( <BEFORE> | <AFTER> | <AROUND> )
                      InputArguments ( <COLON> Identifier
                      InputArguments StatementBlock )
StaticIntroduction ::= ( StaticVariableIntroduction |
                       StaticMethodIntroduction )
StaticVariableIntroduction ::= ( VisibilityDefinition ( EntityType
                             MethodReference EntityDimensions (
                             <ASSIGN> Initializer )? ( <COMMA>
                             Identifier EntityDimensions ( <ASSIGN>
                             Initializer )? )* ) <SEMICOLON> )
StaticMethodIntroduction ::= ( VisibilityDefinition <METHOD>
                           OutputArguments <EQUAL> MethodReference
                           InputArguments StatementBlock )
```

```
StatementBlock ::= ( <OPENBRACE> ( AttributeDeclaration )* (
                   StatementBlock | Statement )* <CLOSEBRACE> )
Statement ::= ( ConditionalStatement | StateMachine | WhileStatement
              | DoWhileStatement | ForStatement | OutputConnection
              <SEMICOLON> | InputConnection <SEMICOLON> |
              MultiAssignment <SEMICOLON> | PreIncrement <SEMICOLON>
              | PreDecrement <SEMICOLON> | PostIncrement <SEMICOLON>
              | PostDecrement <SEMICOLON> | AssignmentStatement
              <SEMICOLON> | Call <SEMICOLON> | NewObject <SEMICOLON>
              | <SEMICOLON> )
ConditionalStatement ::= ( <WHEN> <OPENPARENTHESIS> Expression
                         <CLOSEPARENTHESIS> StatementBlock (
                         <OTHERWISE> StatementBlock )? )
StateMachine ::= ( <STATE> <OPENPARENTHESIS> ReferenceName <COMMA>
                 ReferenceName <CLOSEPARENTHESIS> <OPENBRACE> (
                 TransitionRule )+ <CLOSEBRACE> )
TransitionRule ::= ( Identifier <TRANSITION> Identifier <WHEN>
                   <OPENPARENTHESIS> Expression <CLOSEPARENTHESIS>
                   <SEMICOLON> )
WhileStatement ::= ( <WHILE> <OPENPARENTHESIS> Expression
                   <CLOSEPARENTHESIS> StatementBlock )
DoWhileStatement ::= ( <DO> StatementBlock <WHILE> <OPENPARENTHESIS>
                     Expression <CLOSEPARENTHESIS> <SEMICOLON> )
ForStatement ::= ( <FOR> <OPENPARENTHESIS> ReferenceName <ASSIGN>
                 Range <CLOSEPARENTHESIS> StatementBlock )
OutputConnection ::= ( <CONNECTION> <OPENPARENTHESIS> <STRING_LITERAL>
                     ( "," <STRING_LITERAL> )* <CLOSEPARENTHESIS>
                     <ASSIGN> Expression )
InputConnection ::= ( ReferenceName <ASSIGN> <CONNECTION>
                    <OPENPARENTHESIS> <STRING_LITERAL> ( ","
                    <STRING_LITERAL> )* <CLOSEPARENTHESIS> )
```

```
AssignmentStatement ::= ( ReferenceName <ASSIGN> Expression )
MultiAssignment ::= ( <OPENBRACKET> ParameterReturnList
                    <CLOSEBRACKET> <ASSIGN> Call )
ParameterReturnList ::= ( ReferenceName ( "," ReferenceName )* )
Call ::= ( ReferenceName <OPENPARENTHESIS> ParameterAssignmentList
         <CLOSEPARENTHESIS> )
ParameterAssignmentList ::= ( Expression ( "," Expression )* )
NewObject ::= ( <NEW> ReferenceName <OPENPARENTHESIS>
              ParameterAssignmentList <CLOSEPARENTHESIS> )
PreIncrement ::= <INCREMENT> ReferenceName
PreDecrement ::= <DECREMENT> ReferenceName
PostIncrement ::= ReferenceName <INCREMENT>
PostDecrement ::= ReferenceName <DECREMENT>
Range ::= Expression <ELLIPSE> Expression ( <COMMA> StatementBlock )?
Expression ::= ( InclusiveOr ( <WHEN> <OPENPARENTHESIS> InclusiveOr
               <CLOSEPARENTHESIS> ( InclusiveOr <WHEN>
               <OPENPARENTHESIS> InclusiveOr <CLOSEPARENTHESIS> )*
               ( InclusiveOr <OTHERWISE> )? )? )
InclusiveOr ::= ExclusiveOr ( <OR> InclusiveOr )?
ExclusiveOr ::= And ( <XOR> ExclusiveOr )?
And ::= Equality ( <AND> And )?
Equality ::= Relational ( <EQUALITY> Equality )?
Relational ::= Shift ( <RELATION> Relational )?
Shift ::= Add ( <SHIFT> Shift )?
Add ::= Multiply ( <ADD> Add )?
Multiply ::= Unary ( <MULTIPLY> Multiply )?
Unary ::= ( ( <NOT> | <ADD> )? Primary )
Primary ::= ( <OPENPARENTHESIS> Expression <CLOSEPARENTHESIS>
            | Constant | NewObject | PreIncrement | PreDecrement
            | ReferenceName | PostIncrement | PostDecrement | Call )
            ( <OPENANGLE> Range <CLOSEANGLE> )?
```

```
ReferenceName ::= Identifier ( <OPENBRACKET> ( Range | Expression )
                  <CLOSEBRACKET> )* ( <DOT> Identifier ( <OPENBRACKET>
                  ( Range | Expression ) <CLOSEBRACKET> )* )*
Identifier ::= ( <IDENTIFIER> )
Constant ::= ( <INTEGER_LITERAL> | <STRING_LITERAL> |
             <BOOLEAN_LITERAL> )
MethodReference ::= ( <IDENTIFIER> ) ( <DOT> <IDENTIFIER> )+
```

Appendix B

VHDL Tutorial

This section describes basic syntax of VHDL language and is basically a tutorial covering how to programme in VHDL. However, this is a very important section because the back-end of this project produces VHDL language; hence, deep understanding of VHDL language is required. In Chapter 7, the back-end generation, a few references to this tutorial were made rather than describing the syntax of VHDL again.

B.1 Entity & Architecture

VHDL itself has two major parts, known as *entity* and *architecture*. The *entity* defines the input and output ports. Note that the *entity* name should be the VHDL filename.

The basic syntax of an *entity* is as follows:

```
ENTITY example1 IS
  PORT (x1, x2, x3 : IN BIT;
        f          : OUT BIT);
END example1
```

The bit type used has either high (1) or low (0). More data types are explained in a further section. The above example generates a block that has three input pins and one output pin. We have created the outer shell of a logic block as shown in Figure B.1.

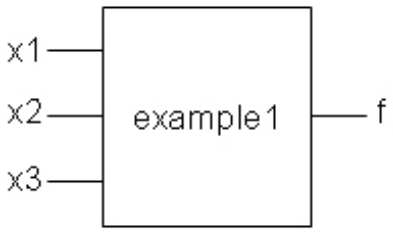


Figure B.1: Entity

Now we need to define the *architecture* that tells how the output is generated from the inputs.

```
ARCHITECTURE LogicFunc OF example1 IS
BEGIN
   f <= (x1 AND x2) OR (NOT x2 AND x3);
END LogicFunc
```

This will create a logic block as shown in Figure B.2 inside our previous empty block, Figure B.1. By combining the *entity* and the *architecture* we have successfully built a working logic function in VHDL.

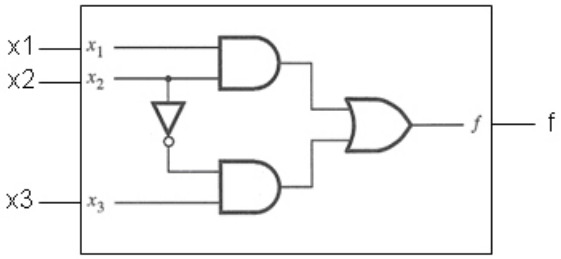


Figure B.2: Logic block

B.2 Library, Package & Component

B.2.1 Library

Library provides a way to enhance the structure of your programme and provide reusability. It is followed by *USE* statements. They are like '`#include <stdio.h>`' in *C* or '`import java.io.*`' in *Java*. The main library that we will be using is the 'IEEE' library.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

The above two *USE* statements are similar to *Java import* statements discussed earlier. The 'all' at the end of *USE* statements is the same as '$*$' in *Java*. You can specify each module but 'all' will simply include everything.

Package

Package gives modularity. *Entity* declarations commonly used can be made as a package. They are similar to including a header file in *C* or making an abstract class in *Java*.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY fulladd IS
  PORT ( Cin, x, y : IN  STD_LOGIC;
           S, Cout : OUT STD_LOGIC);
END fulladd;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
  S <= x XOR y XOR Cin;
  Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y);
END LogicFunc;
```

The above example is a VHDL code for the full-adder. Say that now we want to build a four-bit adder. Then we do not have to create an adder again. We can simply bring the one-bit adder (full adder) and reuse it four times to make a four-bit adder. Here we have two choices — the structural model method or the package method. We will first look at the structural model method.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY adder4 IS
  PORT (Cin            : IN  STD_LOGIC;
        x3, x2, x1, x0 : IN  STD_LOGIC;
        y3, y2, y1, y0 : IN  STD_LOGIC;
        s3, s2, s1, s0 : OUT STD_LOGIC;
        Cout           : OUT STD_LOGIC);
```

```
END adder4;

ARCHITECTURE Structure OF adder4 IS
  SIGNAL c1, c2, c3 : STD_LOGIC;
  COMPONENT fulladd
    PORT (Cin, x, y : IN  STD_LOGIC;
          s, Cout  : OUT STD_LOGIC);
  END COMPONENT;
BEGIN
  stage0: fulladd PORT MAP(Cin, x0, y0, s0, c1);
  stage1: fulladd PORT MAP(C1, x1, y1, s1, c2);
  stage2: fulladd PORT MAP(c2, x2, y2, s2, c3);
  stage3: fulladd PORT MAP(Cin=>c3, Cout=>Cout, x=>x3, y=>y3, s=>s3);
END Structure;
```

Or alternatively, we can use a package to include the 'fulladd' package and import it into the four-bit adder. We first need to create a package.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE fulladd_package IS
  COMPONENT fulladd
    PORT (Cin, x, y  : IN  STD_LOGIC;
          s , C out : OUT STD_LOGIC);
  END COMPONENT;
END fulladd_package;
```

This code can be stored in a separate VHDL source code file or it can be included in the same source code file used to store the code for the 'fulladd' *entity* shown earlier. Note that the component name 'fulladd' must be the same as the *entity* name if this package is saved in a separate file. This can be imported by use of *component* described in the next subsection.

B.2.2 Component

When you define your own package, it is used in conjunction with *component*. Any VHDL *entity* can then use the 'fulladd' component as a subcircuit by making use of the 'fulladd_package' package. The package is accessed using the two statements as follows:

```
LIBRARY work;
USE work.fulladd_package.all;
```

The library named 'work' represents the working directory where the VHDL code that defines the package is stored. This statement is not necessary because the VHDL compiler always has access to the working directory but the *USE* statement is necessary.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.fulladd_package.all;

ENTITY adder4 IS
  PORT (Cin            : IN  STD_LOGIC;
        x3, x2, x1, x0 : IN  STD_LOGIC;
        y3, y2, y1, y0 : IN  STD_LOGIC;
        s3, s2, s1, s0 : OUT STD_LOGIC;
        Cout           : OUT STD_LOGIC);
END adder4;

ARCHITECTURE Structure OF adder4 IS
  SIGNAL c1, c2, c3: STD_LOGIC;
BEGIN
  stage0: fulladd PORT MAP (Cin, x0, y0, s0, c1);
  stage1: fulladd PORT MAP (c1, x1, y1, s1, c2);
  stage2: fulladd PORT MAP (c2, x2, y2, s2, c3);
  stage3: fulladd PORT MAP (Cin=>c3, Cout=>Cout, x=>x3, y=>y3, s=>s3);
END Structure;
```

B.3 Data types

B.3.1 STD_LOGIC

We first looked at a data type called a *BIT*. The *BIT* has only two states. Either high or low, 0 or 1, respectively. This cannot provide all of the states that can actually exist in hardware. Hence *STD_LOGIC* has been developed. It provides a total of nine states, but four are commonly used — 0, 1, Z and X. In order to use *STD_LOGIC*, '`ieee.std_logic_1164.all`' must be included. This is similar to needing to include '`stdio.h`' library to use '`printf`' in *C* and import '`java.io.*`' to use '`System.out.println`' in *Java*. The available types are shown in Table B.1.

U	Uninitialized
X	Unknown
0	Low
1	High
Z	High impedance
W	Weak unknown
L	Weak low
H	Weak high
–	Don't care

Table B.1: Available types for STD_LOGIC

B.3.2 STD_LOGIC_VECTOR

STD_LOGIC can only store a single bit; therefore, *STD_LOGIC_VECTOR* is required to store multiple bits.

```
x : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
y : OUT STD_LOGIC_VECTOR(0 TO 3);
```

The top, `x` statement defines the four bit input port with MSB (Most Significant Bit) first whereas the bottom, `y` statement defines the four bit output port with LSB (Least Significant Bit) first.

B.3.3 Signal

The outputs are generated by manipulating the inputs but it is not always that simple. We need to have some intermediate values. Think about a logic block that contains two logic blocks. Then you need a wire connecting those two logic blocks and the wire will have some intermediate values. The *signal* is used in VHDL to store the intermediate values. They are declared within the *architecture*.

```
SIGNAL Cout : STD_LOGIC;
```

Since the *signal* can be read or written, we do not need to specify whether they are 'IN' or 'OUT'. 'Clock' is also one type of *signal*. *Signals* can have many different attributes associated with their signal and one of the commonly used attributes is as follows:

```
signal_name'event
```

This returns the boolean value *true* if an event occurred otherwise, *false*.

```
IF (CLOCK'event and CLOCK='1') THEN
```

This checks for the arrival of positive clock edge triggers and does the appropriate action in the next statement.

B.4 Functions and Procedures

Like a function in a high-level programming language such as *C*, VHDL functions also take some arguments and return a result. Functions and procedures are the sub-program in a program.

Take a look at this simple example. How the function is used and what it does should be self-explanatory.

```
ARCHITECTURE Inhibit_archf OF Inhibit IS

FUNCTION ButNot (A, B: BIT) RETURN BIT IS
BEGIN
    IF B = '0' THEN RETURN A;
    ELSE RETURN '0';
    END IF;
END ButNot;

BEGIN
    Z <= ButNot(X, Y);
END Inhibit_archf
```

A procedure is slightly different from a function. The function always gives the result in the return value but a procedure does not require a return value. Therefore in most cases, a function is used in an expression where a value is assigned to another variable or signal (i.e. encapsulates statements to produce a result), but a procedure is used in a statement (i.e. generalization of a collection of statements). However, procedure can also be used to return a value; in fact, more than one value can be returned by using the arguments as 'OUT' or 'INOUT' type. It is similar to functions having pointer arguments in *C*.

```
PROCEDURE count (inc: IN Boolean; big: OUT BIT; num: INOUT integer) IS
-- type, variable, constant, subprogram declarations
BEGIN
-- sequential statements
  IF inc THEN
    num := num + 1;
  END IF;
  IF num > 111 THEN
    big := '1';
  ELSE
    big := '0';
  END IF;
END;
```

The next example shows a way to read different numbers of parameters with the same procedure name, like *overloading* in *Java*.

```
PROCEDURE read(l: INOUT line; value: OUT signed) IS
-- variable
BEGIN
-- body
END

PROCEDURE read(l: INOUT line; value: OUT signed; good: OUT Boolean) IS
-- variable
BEGIN
-- body
END
```

B.5 Process

A *process* is the main part in behavoural modelling. I have mentioned earlier that the VHDL statements are executed concurrently. However, the statements within a *process* are executed sequentially. You can think of a process as a *thread* in *Java*. A process takes some signals as input arguments and changes the value within the *process*.

An example of a positive edge-triggered D flip-flop is shown below.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff is
  PORT (data, clk : IN  STD_LOGIC;
```

```
          q      : OUT STD_LOGIC);
END dff;

ARCHITECTURE behav OF dff IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF (clk'event and clk='1') THEN
      q <= data;
    END IF;
  END PROCESS;
END behav;
```

A *process* reads and writes the signals and values of the input and output ports to communicate with the rest of the *architecture*.

B.6 Basic commands

Most of the basic commands are similar to other programming languages such as *C, Pascal, Java* and *MATLAB*. Also, like any other programming language, there are several different ways to achieve the same task.

B.6.1 IF ... THEN

```
IF condition THEN
  sequential statements
  [ELSIF condition THEN
    sequential statements]
  [else
    sequential statements]
END IF;
```

B.6.2 CASE

```
CASE expression IS
  WHEN choices =>
    sequential statements
  WHEN choices =>
    sequential statements
  [WHEN others =>
    sequential statements]
END CASE;
```

B.6.3 WHILE

```
[loop_label :] WHILE condition LOOP
  sequential statements
  [NEXT [label] [WHEN condition];
  [EXIT [label] [WHEN condition];
END LOOP [loop_label];
```

The condition of the loop is tested before each iteration, including the first iteration. If the condition is *false*, then the loop is terminated.

B.6.4 FOR

```
[loop_label :] FOR identifier IN range LOOP
  sequential statements
  [NEXT [label] [WHEN condition];
  [EXIT [label] [WHEN condition];
END LOOP [loop_label];
```

B.7 Designing FSM

State machines are the most commonly used method of processing the mode. Even a vending machine will simply have some states, for example,

```
READY -> MONEY IN -> MORE MONEY IN -> GIVE COKE -> GIVE CHANGE
```

Let us look at this simple diagram of a state machine, as shown in Figure B.3. Designing this will involve most of the things we have covered in this tutorial.

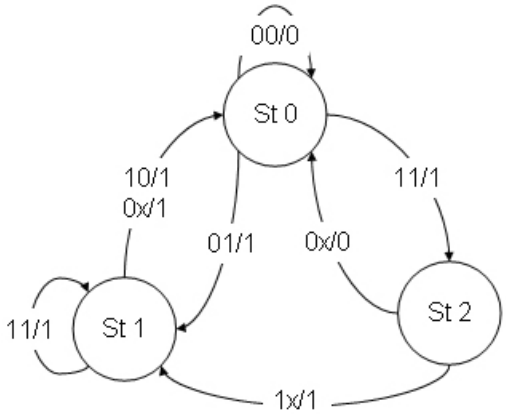


Figure B.3: FSM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY FSMeg IS
  PORT (clk, reset : IN  STD_LOGIC;
        data_out   : OUT STD_LOGIC;
        data_in    : IN  STD_LOGIC_VECTOR(1 DOWNTO 0));
END FSMeg;

ARCHITECTURE behav OF FSMeg IS
  TYPE states IS (ST0, ST1, ST2);
  SIGNAL pre_state, next_state : state_values;
BEGIN
  statereg : process(clk, reset)
  BEGIN
    IF (reset = '0') THEN
      pre_state <= ST0;
    ELSIF (clk'event and clk='1') THEN
      pre_state <= next_state;
    END IF;
  END process statereg;

  fsm: process(pre_state, data in)
  BEGIN
    CASE pre_state IS
      WHEN ST0 =>
        CASE data_in IS
          WHEN "00" => next_state <= ST0;
          WHEN "01" => next_state <= st1;
          WHEN "10" => next_state <= st2;
          WHEN others => null;
        END CASE;
      WHEN ST1 =>
        CASE data_in IS
          WHEN "10" => next_state <= ST0;
          WHEN "11" => next_state <= ST1;
          WHEN others => next_state <= ST0;
        END CASE;
      WHEN ST2 =>
        CASE data_in IS
          WHEN "00" => next_state <= ST0;
          WHEN "01" => next_state <= ST0;
          WHEN "10" => next_state <= ST1;
          WHEN "11" => enxt_state <= ST1;
        END CASE;
      WHEN others => next_state <= ST0;
    END CASE;
```

```
  END process fsm;

  outputs: process(pre_state, data_in)
  BEGIN
    CASE pre_state IS
      WHEN ST0 =>
        CASE data_in IS
          WHEN "00" => data_out <= '0';
          WHEN others => data_out <= '1';
        END CASE;
      WHEN ST1 =>
        data_out <= '1';
      WHEN ST2 =>
        CASE data_in IS
          WHEN "00" => data_out <= '0';
          WHEN "01" => data_out <= '0';
          WHEN "10" => data_out <= '1';
          WHEN "11" => data_out <= '1';
        END CASE;
      WHEN others => data_out <= '0';
    END CASE;
  END process outputs;

END FSMeg;
```

The first part is easy: designing the inputs and outputs in the *entity*. Within the *architecture*, we need to make three state-type variables and a signal that stores the previous state and the next state. The states are changed only on clock events. Now we need to design how the states are changed. Note that we have divided the process into three. One controls and changes the actual state, one processes the FSM, how the states are changed and under what condition, and the other does the output generation. Although we can implement this in one process, it is a good practice to divide the process in this way.

Appendix C

AspectJ Tutorial

In ADH, we provide aspect features that are modelled from *AspectJ* [2]. I have also used *AspectJ* as a primary language to implement the compiler. Hence, I am presenting a brief summary of *AspectJ* in the following section.

C.1 What is AspectJ?

AspectJ is the general-purpose aspect-oriented programming extension to Java. It was first introduced in 1998 and is still in development. At the time of writing the tutorial, early 2005, the current version 1.2.1 was found on "http://eclipse.org/aspectj".

AOP is still new to most software developers and new to myself. There are not many books available at the time of writing. Therefore this section is heavily based on a book [53] stated in the bibliography.

C.2 What makes AOP interesting? & When to use it?

There are major examples that are used to describe the advantages of AOP. The major one is a Logging example. I will use diagrams to help understand when to use AOP. This is like a design pattern. "When you see something that fits into a certain category, then do it this way." is what a design pattern basically teaches you.

In Figure C.1, the left-hand boxes represent objects from a banking system. Each client module will contain a part for logging. The codes are not the same but all of them do the same task — logging. If they are taken out from that object and can be implemented as a single object, the other codes will look cleaner and hence easier to understand and maintain. However, in OOP, this is not possible. They should reside in each module. What AOP does is build a bridge between the logging module and other modules, called logging aspect. It uses a technique called weaving.

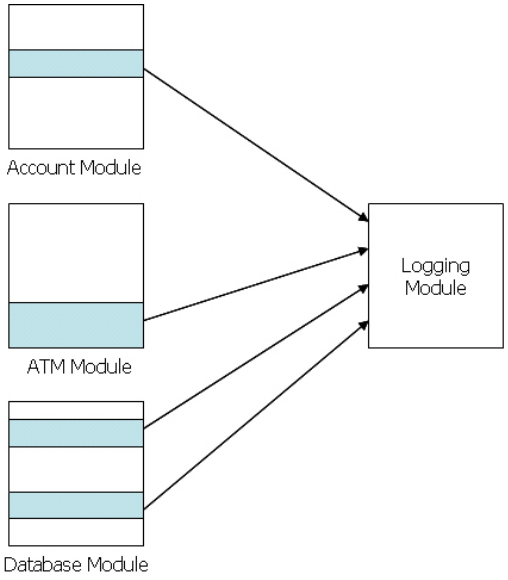


Figure C.1: Logging example

C.3 Syntax basics

Before we go into more detail, take a look at how AspectJ looks.

```
class HelloWorld {
  public void greeting() {
    System.out.println("Hello World.");
  }

  public static void main(String args[]) {
    HelloWorld me = new HelloWorld();
    me.greeting();
  }
}
```

This is a normal *Java* file that we have all come across when we first started programming. The compile and execution process will give the following output.

```
> javac HelloWorld.java
> java HelloWorld

Hello World.
```

In addition to this code, we make a 'TestAspect' that is cut into the 'Test' class.

```
public aspect LogAspect {
  pointcut outputLog() : call(public void HelloWorld.greeting());
  before() : outputLog() {
    System.out.println("Before Call");
  }
}
```

You will probably be able to guess what the output will produce.

```
> ajc LogAspect.aj HelloWorld.java
> java HelloWorld

Before Call
Hello World.
```

Now we will look at how this code is working. You basically need to know what *pointcut* is, and what *advice* is.

C.4 Join Point

A *Join point* is a well-defined execution point in a system. A call to a method is a *join point.*

```
Public class Account {
  ...
  void credit (float amount) {
    balance = balance + amount;
  }
}
```

The *join points* in the Account class include the execution of the credit() method and the access to the 'balance' instance member.

C.5 Pointcut

Pointcuts capture *join points* in the program flow. Once you capture the *join points*, you can specify weaving rules involving those *join points*. We can write a *pointcut* that will capture the execution of the 'credit()' method in the 'Account' class shown earlier.

```
execution(void Account.credit(float))
```

To understand the difference between a *join point* and *pointcut*, think of *pointcuts* as specifying the weaving rules and join points as situations satisfying those rules.

The full syntax with an example is illustrated in Figure C.2.

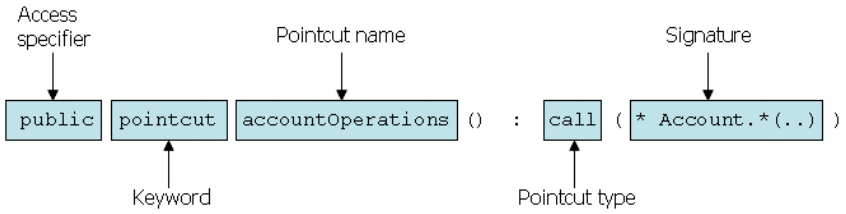


Figure C.2: AspectJ Syntex

Within the signature, the '$*$' denotes any number of characters except the period. The '..' denotes any number of characters including any number of periods and '+' denotes any subclass or subinterface of a given type.

The above particular example says, "All methods in the Account class taking any number and type of arguments and returning any type. This will even match methods with private access."

C.6 Advice

Basically, there are three *advices* that can be applied — *before*, *after* and *around*. *Before* advice executes prior to the *join point*. *After* advice executes following the *join point*. *Around* advice causes a procedure to bypass some lines of execution. This *advice* is special in that it has the ability to bypass execution, continue the original execution, or cause execution with an altered context.

Using the earlier *pointcut*, we can write advice that will print a message before the execution of the 'credit()' method in the 'Account' class.

```
before() : execution(void Account.credit(float)) {
    System.out.println("About to perform credit operation");
}
```

Pointcuts and *advice* together form the dynamic crosscutting rules. While the *pointcuts* identify the required *join points*, the *advice* completes the picture by providing the actions that will occur at the *join points*.

Advice implementation often requires the access to data at the *join point*. Figure C.3 is an example of passing an executing object and an argument context from the *join point* to the *advice* body. *AspectJ* provides some keywords — *this*, *target*, and *args* — at *pointcut* to capture and collect the context.

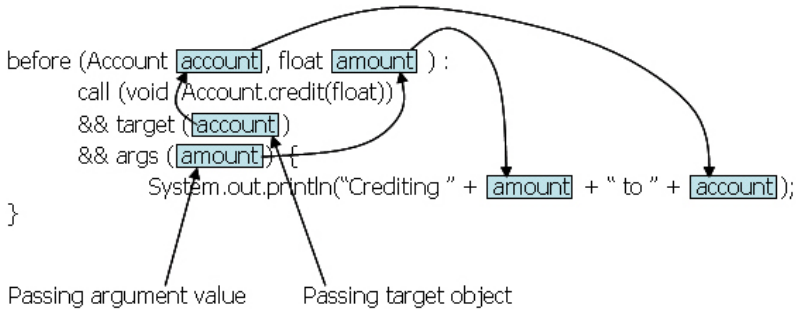


Figure C.3: Passing context from a *join point* to *advice* [53]

C.7 Aspect

An *aspect* is the central unit of *AspectJ*, like *class* is the central unit in *Java*. It contains the code for weaving rules for both dynamic and static crosscutting. We include all of the three major declarations into an aspect and the codes are similar to a class.

```
public aspect ExampleAspect {
   before() : execution(void Account.credit(float)) {
      System.out.println("About to perform credit operation");
   }

   pointcut updateAccount(String s) :
      call(public String MyTrans.updateAcc(String) && args(s));
}
```

In this chapter, a very basic tutorial of *AspectJ* is provided. For more details and latest updates, you should refer to the textbooks [31, 53, 57] and the online resources [2].

Appendix D

Implementation techniques

There are number of notable programming techniques that can be applied to the compiler implementation such as the design patterns. The design patterns are general repeatable solutions to commonly occurring problems. The patterns are not complete codes that can be used; they are the templates and can be used to solve many different cases. The important patterns and the ones I have used are described in one of the subsections.

The main feature in ADH is aspect-oriented support. The compiler implementation itself is also done with the aspect-oriented language, *AspectJ*. This chapter also describes some particular implementation techniques using *AspectJ* for compiler implementation; however, it may require the reader to refer to Chapters 5 and 6.

D.1 Design Patterns

The design patterns were first proposed by the "Gang of Four" [29] and are commonly named the GoF design patterns. More patterns have been developed by other researchers since the GoF design patterns [32,63]. The patterns contain some of the useful design methodologies that arose when developing software programs. The initial patterns and the ones we have used include the following:

- Abstract factory pattern
- Singleton pattern

- Strategy pattern
- Visitor pattern

The abstract factory pattern and singleton patterns are called creational patterns as they help in the creation of an object. Both patterns are used in the 'symbols' package, and they create and manage the symbols and the symbol table. The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme. The 'singleton pattern' [29] is designed to restrict instantiation of a class to one object. It basically prevents the multiple instance of an object where there should not be more than one instance. In a compiler, there should only be one symbol table; hence, applying this pattern prevents mis-usage of a symbol table creation.

The 'strategy pattern' [29] and the 'visitor pattern' [32] fall into behavioral patterns. They are used to organize, manage, and combine behaviours of the objects. In this project, the 'visitor pattern' is the most notable one; therefore, I will introduce the 'visitor pattern' in more detail here. The details of the other patterns can be found on many books [29, 32, 63].

D.1.1 Visitor pattern

The 'visitor pattern' helps in many ways. The main purpose is to provide new operations on a set of related classes without actually altering the classes involved. A way to extend a class hierarchy's behaviour is to add methods that provide the behaviour you need. This functionality is separated for readability and maintenance reasons.

It is used to traverse a tree structure efficiently because the 'visitor pattern' specifies how iteration occurs over the object structure. The basic idea is that each object has an *accept* method that takes a visitor object as an argument. The visitor is an interface that has a different *visit()* method for each element class. The *accept()* method of an element class calls back the *visit()* method for its class. As a result, calling a separate concrete visitor class is done in order to carry out certain operations. This calling mechanism is shown in Figure D.1, and Figure D.2 shows

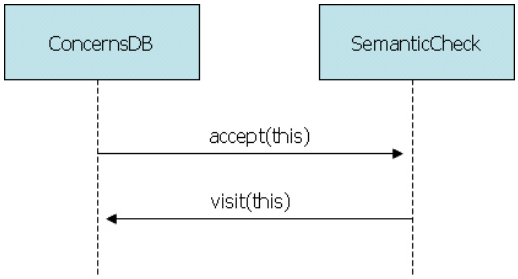


Figure D.1: The calling mechanism of a visitor pattern

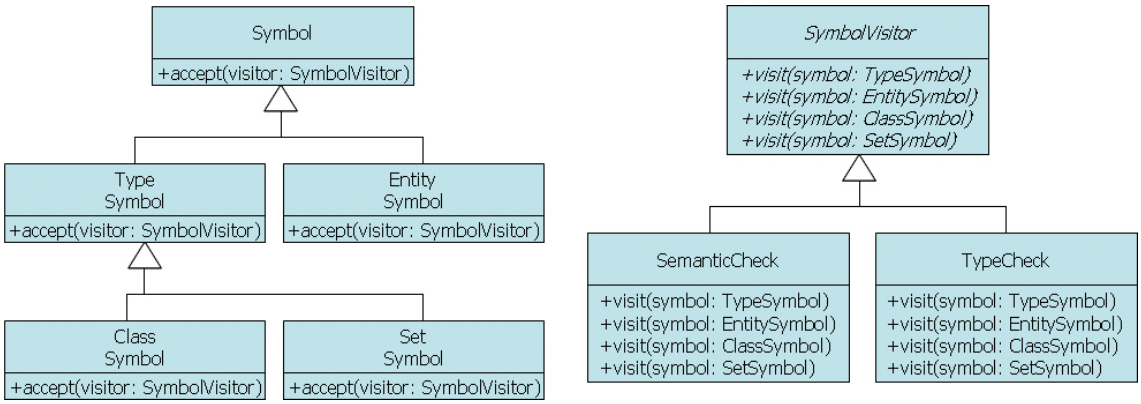


Figure D.2: Partial UML diagram showing the implementation of the visitor pattern to the 'symbols' package

a partial UML (Unified Modelling Language) [28] diagram for a 'symbols' package that implements 'visitor pattern'.

The visitor object has only one principal function with the name *visit()*; therefore, the visitor can be readily identified as a potential function object. Likewise, the *accept()* function can be identified as a function applicator, a mapper, which knows how to traverse a particular type of object and apply a function to its elements.

D.1.2 Aspect visitor pattern

The 'ordinary object-oriented visitor pattern' (ordinary visitor pattern) is a little awkward, due to the double calling of the functions, previously shown in Figure D.1. It is argued that this pattern is unnatural and contains some problems in terms of software engineering. Each class requires the same *visit()* method that is identical but they cannot be moved to the higher hierarchy due to the parameter being the object itself (i.e. "`visit(this)`"). The applicators require the implementation of

the *visit()* method for all of the classes regardless of their usage. The body of the method that does not access the object is usually "`return null;`" and it is possible to mis-use this method body. This also increases the *coupling* and reduces the *cohesion*.

Using the features in AOP, the 'ordinary visitor pattern' can be re-written as an 'aspect visitor pattern', which is cleaner to code and understand. It totally removes the *visit* and *accept* method and their calling mechanisms.

Recall the earlier 'ordinary visitor pattern'. There is a major problem apart from the software engineering issue in that the calling mechanism is awkward. From this project, symbols hierarchy can be shown in Figure 5.3 with some 'visitor pattern' operations.

Consider the following example; see Figure D.3. The symbol hierarchy could be modified to add some more functionality. For example, a new symbol, say 'FSMSymbol', could be added to the symbol hierarchy. A developer of a symbols hierarchy may have made changes and added the visitor support but the developer on the supporting visitor pattern may not realise or is not able to make changes in time. However, it will still compile even though the visitor patterns do not support the additional symbol that may cause problems in runtime. The runtime-error will occur when the 'visitor pattern' finds an un-recognized symbol, as shown in Figure D.3. Aspect visitor patterns find such problems in compile-time and produce appropriate error messages therefore preventing a runtime exception.

D.2 Use of AspectJ

This section describes some of the implementation techniques using *AspectJ* in the intermediate-stage development.

D.2.1 Static Cross-cutting usage

Intermediate translation is written in *AspectJ*. The translation requires access to each symbol in the symbols' hierarchy and also it requires the access to each node of the parse-tree. Therefore, without the aspect features, all of the intermediate

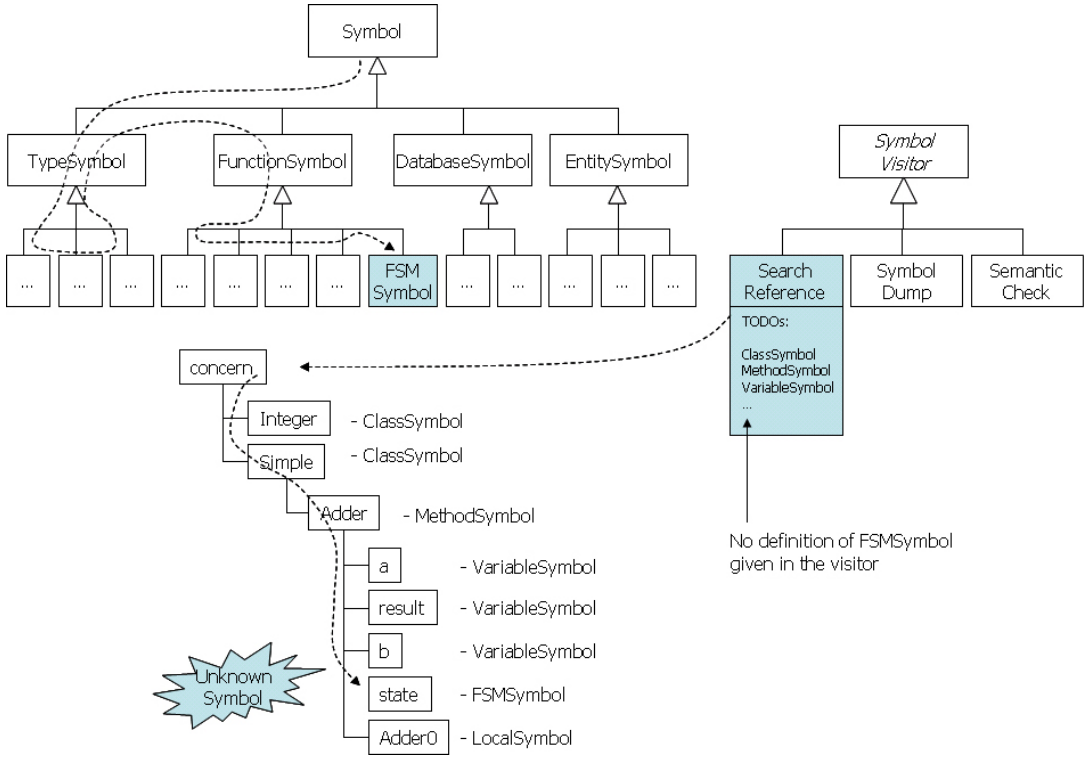


Figure D.3: A problem that might arise from the use of ordinary visitor pattern

translation codes must be split and written in a method for each symbol and each node. This is similar to the logging example described in Appendix C.

As described earlier, this helps making 'low-coupling' and 'high-cohesion' of the codes that is well known by the software engineers. Figure D.4 just illustrates parts of this project using static cross-cutting.

D.2.2 Dynamic Cross-cutting usage

The optimisation techniques are written in dynamic features of *AspectJ*. The optimisation codes are written after the completion of the intermediate translation code. The optimisation occurs depending on the dataflow analysis and again, they occur at the symbol. Using the dynamic cross-cutting features, *percflow* provided in *AspectJ*, an object can automatically be created and bound to the correct symbol as shown in Figure D.5.

The life-time of this *aspect* lies with the life-time of the associated object. Therefore, if the calculated values are necessary for further usage, then they must be copied

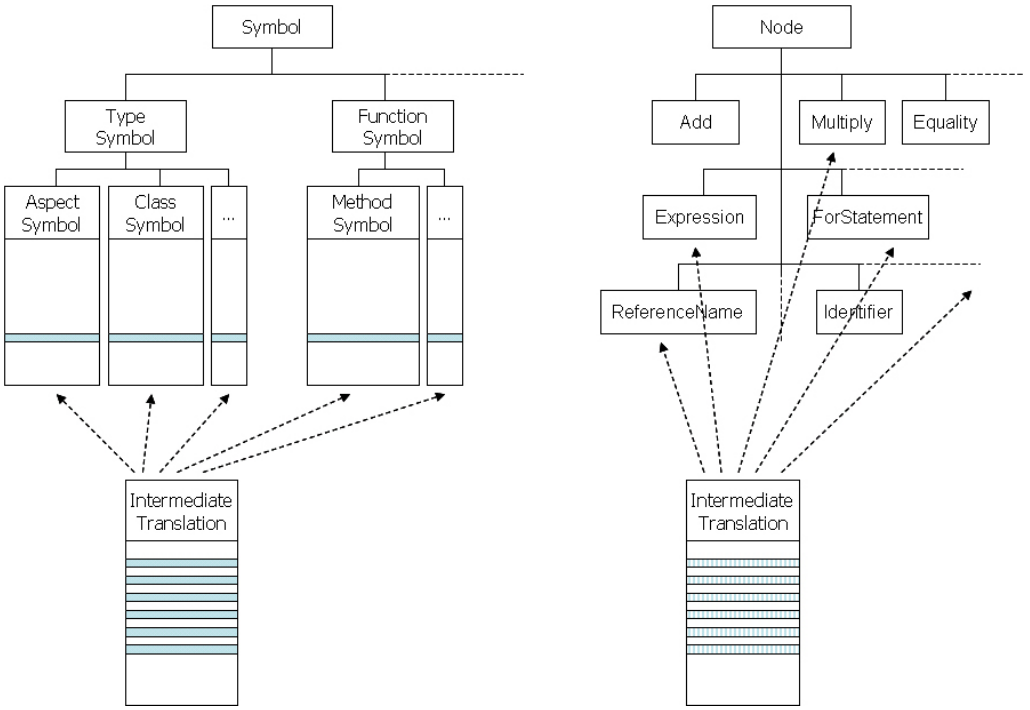


Figure D.4: Static aspect usage in the intermediate stage

to another object.

The figure illustrates that the 'SSAForm' calls and uses many different types of symbols and the automatic instantiation of the optimisation modules such as 'generateDominatorTree'. It gets bound and does its job at the correct function symbol. The information is used for the construction of the 'SSAform' and disappears at the end.

This dynamic aspect usage can also help in the use of the 'cache' approach for the entire optimisation code that is planned to be changed. Currently, the optimisation code is written sequentially by the steps given earlier in the section on page 48. There is a standard way for 'cache' implementation using *AspectJ*. This eliminates the sequential order that the developer sets. Basically it is closer to the OO design in giving instructions to the object, "Do the optimisation", then finding whether the dominator tree is calculated and the dominator frontier is calculated etc. If they do not exist in its 'cache', it does calculate and store in its 'cache'.

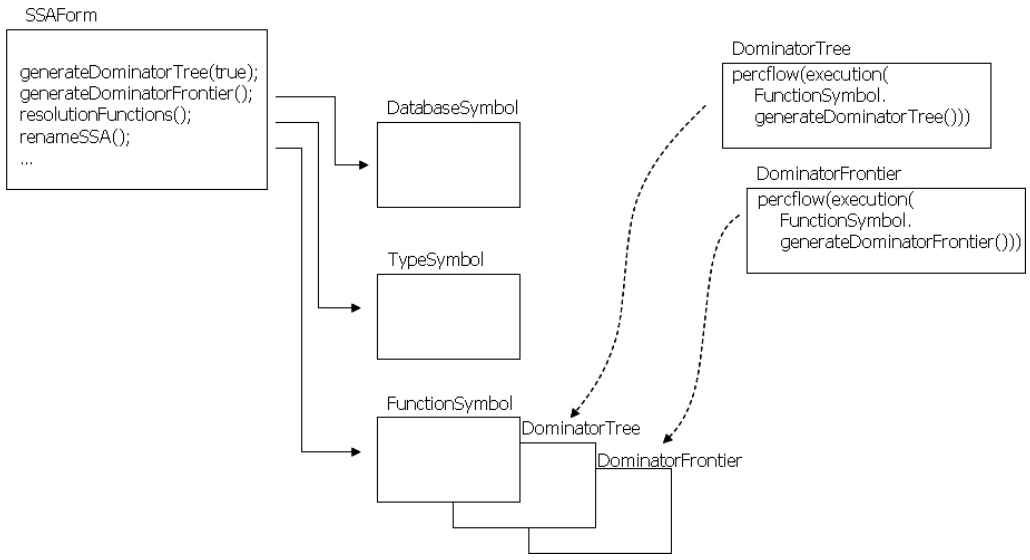


Figure D.5: Dynamic aspect usage

D.2.3 Java5 annotation

Java5 [48] includes a new feature called annotation. Annotations are similar to the comments on a program but can be read by the *Java VM* (Virtual Machine) if required. Figure D.6 shows how the developer sets the policy that the annotation is to be read on run-time and we have set that the default value of 'BlockVisitor' is one, which means each node can only be visited once. The usage of block visitor and the control of the visit is explained shortly.

```
1:  @Retention(RetentionPolicy.RUNTIME)
2:  public @interface BlockVisitor
3:  {
4:    int value() default 1;
5:  }
```

Figure D.6: Java5 Annotation that can be put on to Java VM

In our project, particularly on optimisation techniques, the basic-block nodes are visited multiple times as shown in Figure D.7, and there is no method of tracking the number of visits to the node for different applications. In the figure, on the right-hand side, it shows that each operation — renameSSA, subExpressionReduction and constrantReduction — requires different numbers of visits to the same basic blocks. The visit information can be tracked and stored and we can even control

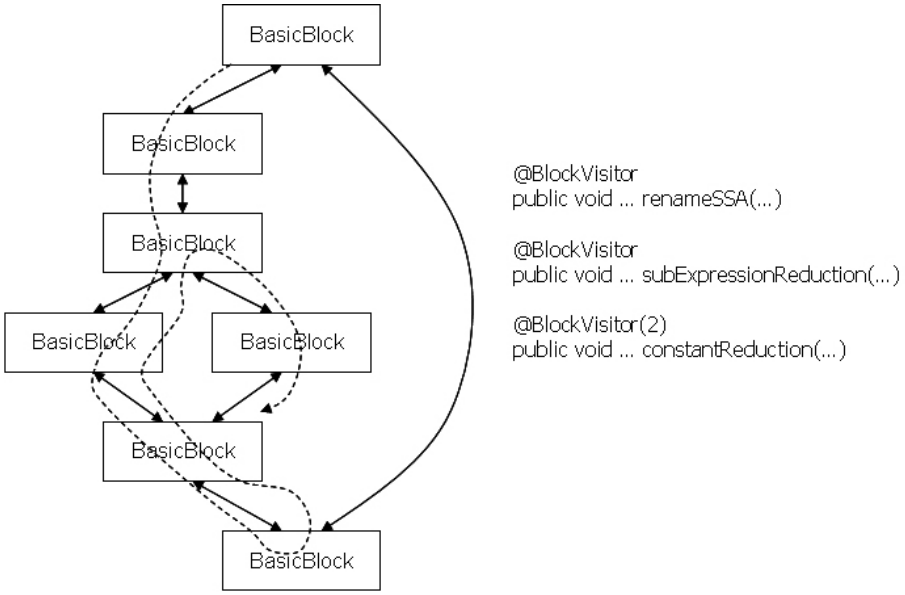


Figure D.7: Different algorithms require different numbers of visits to each node

the number of visits to each node using the annotation and help of *AspectJ*.

In Figure D.8, the first line shows that the number of visits to this node is set at two. Figure D.9 shows the setting of the *pointcut* when a block visitor annotation is created. Then, at line 5 and 6, it adds a marker (for what operation it is performing now) and a variable (for counting the number of visits). The *advice* it is giving, at line 8 to the end, is to bypass the visit to the current basic block using 'around' when the number of visits exceeds the given value.

```
1:  @BlockVisitor(2)
2:  public void controlGraph.BasicBlock.
3:     constantReduction(FunctionSymbol function)
4:  {
5:    ...
```

Figure D.8: Controlling the number of visits to the node using annotation.

D.3 Summary

Some of the useful and repeatable software implementation techniques are shown in this chapter. These techniques help make the development process and implementation easier, and they can also be extended further in places. More design patterns

```
 1:  pointcut visitor(BasicBlock block, BlockVisitor visitor) :
 2:      execution(@BlockVisitor * *.*(..)) &&
 3:      target(block) && @annotation(visitor);
 4:
 5:  private String   controlGraph.BasicBlock.mark = "";
 6:  private int      controlGraph.BasicBlock.visitCount;
 7:
 8:  Object around(controlGraph.BasicBlock block, BlockVisitor visitor) :
 9:      visitor(block, visitor)
10:  {
11:  ...
```

Figure D.9: The aspect inserted when the basic-block visit counter is required. Code extracts from 'BlockVisitor.aj'

can be studied and applied to this project. Refactoring, which is re-writing of the codes without changing their functionality, is also used many times to modularise the codes and reduces the comments by making the code self-explanatory. There are also aspect-oriented design patterns in some papers [37, 39] that could be applied. The paters present the GoF design patterns in *Java* and *AspectJ* and the notable ones are the *Singleton, Prototype, Memento, Iterator* and *Visitor* patterns.

Bibliography

[1] Alfred V. Aho, Ravi. Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley Pub Co., 1986.

[2] The Eclipse Project AspectJ Community. The AspectJ project at Eclipse.org. http://www.eclipse.org/aspectj/. last visited Mar, 2006.

[3] IEEE1394 Technical Association. 1394 Trade Association: Technology. http://www.1394ta.org/Technology/. last visited Mar, 2006.

[4] G. Auty, A Bainbridge-Smith, A. Dreier, and R. Kirkham. *Truckscan bypass lane avoidance camera system. Tech. rep.* CSIRO, 2002.

[5] G. Auty, P. Corke, P. Dunn, M. Jensen, I. MacIntyre, H. Nguyen, and B. Simons. An image acquistion system for traffic monitoring applications. *SPIE*, 2416(133), 1995.

[6] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[7] Sung Eun Bae and Tadao Takaoka. Algorithms for the Problem of K Maximum Sums and a VLSI Algorithm for the K Maximum Subarrays Problem. In *ISPAN*, pages 247–253, 2004.

[8] Sung Eun Bae and Tadao Takaoka. Improved Algorithms for the k-Maximum Subarray Problem for Small . In *COCOON*, pages 621–631, 2005.

[9] A. Bainbridge-Smith and P. Dunn. Fpgas in computer vision applications. In *Proceedings of the Image and Vision Computering New Zeland 2002 Conference*. University of Auckland, 2002.

[10] Andrew Bainbridge-Smith and Su-Hyun Park. ADH: An Aspect Described Hardware Programming Language. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, FPT 2005*, pages 283–284, 2005.

[11] Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, April 1998.

[12] USB Standards Body. Universal serial bus. http://www.usb.org. last visited Mar, 2006.

[13] Luc Bougé. *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume Lecture Notes in Computer Science 1132, chapter The Data Parallel Programming Model: A Semantic Perspective. Springer, 1996.

[14] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digital Logic with VHDL Design*. McGraw Hill, 2000.

[15] Celoxica. Handel-c language reference manual. Technical report, Celoxica, 1998.

[16] Wesley J. Chun. *Core python programming*. Prentice Hall PTR core series. Prentice Hall, Upper Saddle River, NJ, 2001.

[17] Siobhán Clarke and Robert J. Walker. Towards a standard design language for AOSD. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 113–119, New York, NY, USA, 2002. ACM Press.

[18] David R. Coelho. *The VHDL handbook*. Addison-Wesley, 1989.

[19] George A. Constatinides, Peter Y.K. Cheung, and Wayne Luk. *Synthesis and Optimization of DSP Algorithms*. Kluwer Academic Publishers, 2004.

[20] D. Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2004.

[21] W. Curtis, H. Krasner, V. Shen, and N. Iscoe. On building software process models under the lamppost. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 96–103, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[22] Dennis de Champeaux, Doug Lea, and Penelope Faure. The process of object-oriented design. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 45–62, New York, NY, USA, 1992. ACM Press.

[23] P. Dunn. A configurable logic processor for machine vision. In *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications*, volume 975 of *LCS*. Springer, 1995.

[24] G. Economakos, P. Oikonomakos, and I. Panagopoulos. Behavioral synthesis with systemc. Technical report, Dept. Elec. and Comp. Eng., National Technical Univerity of Greece, 2001.

[25] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[26] C. T. Ewe, P. Y. K. Cheung, and G. A. Constantinides. Dual fixed-point: An efficient alternative to floating-point computation. In *Field Programmable Logic and Applications*, volume Lecture Notes in Computer Science. Springer-Verlag, 2004.

[27] R. Ferguson, D. Pratt, and I. MacIntyre. Automated detection and classification of cracking in road pavements (roadcrack). In *Proceedings of the 19th ARRB Transport Research Conference*. Australian Road Research Board, 1998.

[28] Martin Fowler and Kendall Scott. *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley Pub Co., 2nd edition, 2000.

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[30] GNU. Bison - GNU Project. http://www.gnu.org/software/bison/. last visited Mar, 2006.

[31] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ : aspect-oriented programming in Java*. Wiley, 2003.

[32] Mark Grand. *Patterns in Java*. Wiley, 2nd edition, 1999.

[33] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45, New York, NY, USA, 2004. ACM Press.

[34] Steve Guccione, Delon Levi, and Prasanna Sundararajan. JBits: Java based interface for reconfigurable computing. In *MAPLD International Conference*, 1999.

[35] Steven A. Guccione and Delon Levi. JBits: A Java-Based Interface to FPGA Hardware. http://www.io.com/ guccione/Papers/JBits/JBits.html, last visited Mar, 2006.

[36] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[37] Ouafa Hachani and Daniel Bardou. Using Aspect-Oriented Programming for Design Patterns Implementation. In *Reuse in Object-Oriented Information Systems Design workshop. 8th International Conference on Object-Oriented Information Systems (OOIS 2002).*

[38] Stefan Hanenberg, Robert Hirschfeld, and Rainer Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 46–55, New York, NY, USA, 2004. ACM Press.

[39] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and aspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[40] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*.

[41] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[42] Yoav Hollander, Matthew Morley, and Amos Noy. The e language: A fresh separation of concerns. In *TOOLS '01: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 41, Washington, DC, USA, 2001. IEEE Computer Society.

[43] Scott Hudson, Frank Flannery, and C. Scott Ananian. CUP parser generator for Java. http://www.cs.princeton.edu/ appel/modern/java/CUP/. last visited Mar, 2006.

[44] Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, and Mike Rytting. A CAD Suite for High-Performance FPGA Design. In *In Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–24, April 1999.

[45] Micron Imaging. *Mi-mv13 500-fps, 1.3 megapixel CMOS Image Sensor*. http://www.fast-vision.com/downloads/bgawc4510/MI-MV13salessheet.pdf, 2002. last visited Mar, 2006.

[46] Apple Inc. Firewire. http://www.apple.com/firewire. last visited Mar, 2006.

[47] MathWorks Inc. The MathWorks - MATLAB and Simulink for Technical Computing. http://www.mathworks.com. last visited Mar, 2006.

[48] Sun Microsystems Inc. Java 5: New Features and Enhancements, J2SE 5.0. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html. last visited Mar, 2006.

[49] Sun Microsystems Inc. JavaCC Home - JavaCC is a parser/scanner generator for java. https://javacc.dev.java.net/. last visited Mar, 2006.

[50] Sun Microsystems Inc. JavaCC[tm]: JJTree Reference Documentation. https://javacc.dev.java.net/doc/JJTree.html. last visited Mar, 2006.

[51] Stephen C. Johnson. The Lex & Yacc page. http://dinosaur.compilertools.net/. last visited Mar, 2006.

[52] Israel Koren. *Computer Arithmetic Algorithms*. A K Peters, 2002.

[53] Ramnivas Laddad. *AspectJ in Action - Practial Aspect-Oriented Programming*. Manning, 2003.

[54] Wilf LaLonde, John Pugh, Paul White, and Jean-Pierre Corriveau. Towards unifying analysis, design, and implementation in object-oriented environments. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 563–569. IBM Press, 1993.

[55] A. Lambert, J. Webb, and D. Fraser. A fast intelligent image sensor with application to image restoration. In *International Symposium on Optical Science and Technology*, volume 4792A. SPIE, 2002.

[56] Mikrotron. Miktrotron MC1310, MC1311 High Resolution High Speed Cameras. http://www.turnkey-solutions.com.au/cam_mikrotron_mc131011.htm. last visited Mar, 2006.

[57] Russ Miles. *AspectJ cookbook*. O'Reilly Media, 2005.

[58] Gordon E. Moore. Moore's Law, The Future - Technology & Research at Intel. http://www.intel.com/technology/silicon/mooreslaw/. last visited Mar, 2006.

[59] Jean-Marc Nerson. Applying object-oriented analysis and design. *Commun. ACM*, 35(9):63–74, 1992.

[60] Su-Hyun Park and Andrew Bainbridge-Smith. ADH: Aspect Described Hardware-Description-Language. In *Proceedings of the twelveth Electronics New Zealand Conference, ENZCon05*, pages 69–74, Manukau City, New Zealand, November 2005.

[61] David Pellerin and Douglas Taylor. *VHDL made easy!* Prentice Hall, 1997.

[62] Alexandra Poetter, Jesse Hunter, Cameron Patterson, Peter Athanas, Brent Nelson, and Neil Steiner. JHDLBits: The Merging of Two Worlds. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'2004)*, August 2004.

[63] Alan Shalloway and James R. Trott. *Design patterns explained : a new perspective on object-oriented design.* Addison-Wesley, 2002.

[64] Andrew w. Appel. *Modern compiler implementation in Java.* Cambridge university press, 2nd edition, 2002.